

**Introduction**:

In this section we will learn how to secure our API endpoints using permissions.

* Token Based Authentication
* Adding authentication endpoints
* Registering, logging in, etc.
* Applying permission

**Token based Authentication**:

Token based authentication is,

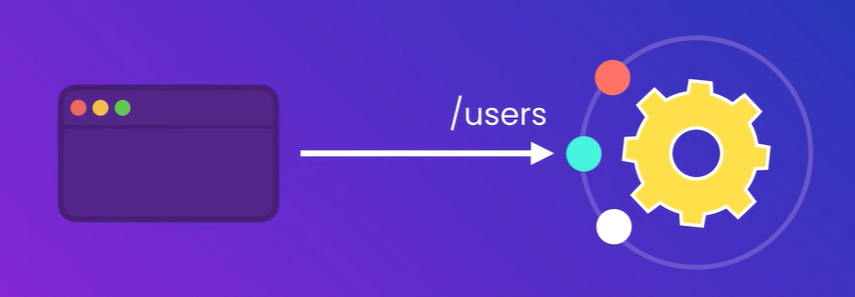


Here is how it works.

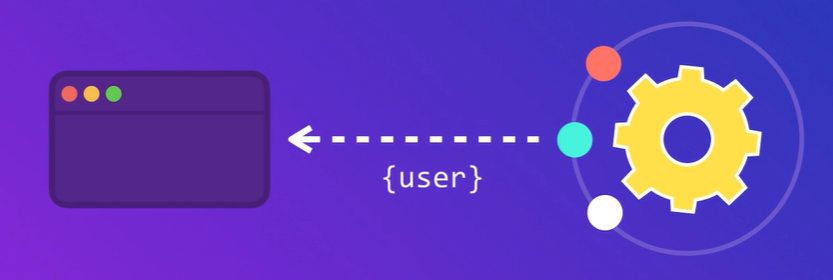
Let’s say a new user is going to use our application, so first they need to register.

*Steps*:

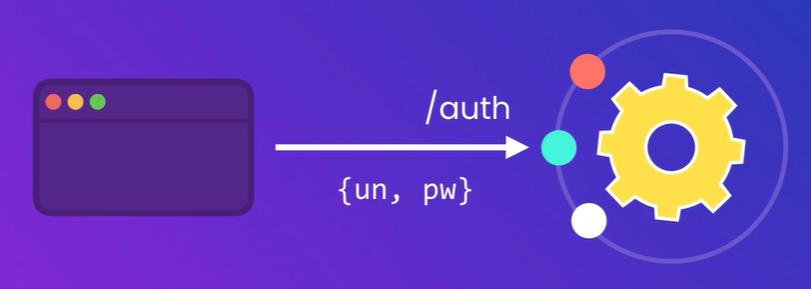
1. On their machine the client app is going to *send a POST request to the /users* end point.



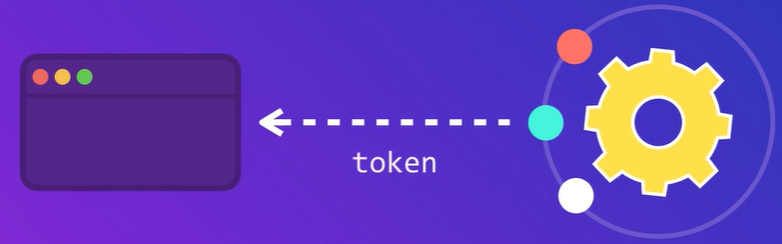
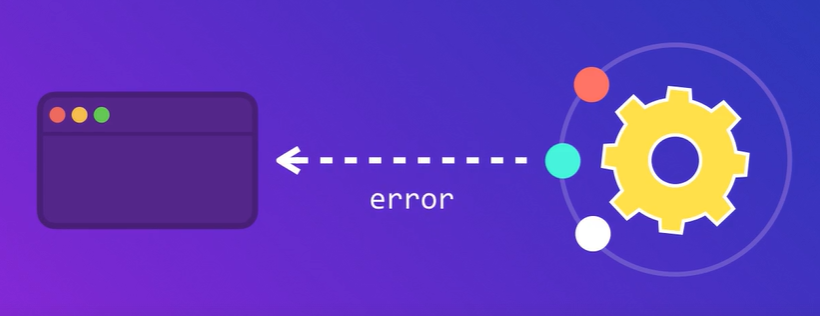
1. On the server, we are going to capture their username, password, email and other fields, then *create a new account for them*.



1. Next the user needs to login, so the client app needs to *send a POST request to the* ***authentication endpoint*** and it should pass the user credentials meaning username and password to the server.



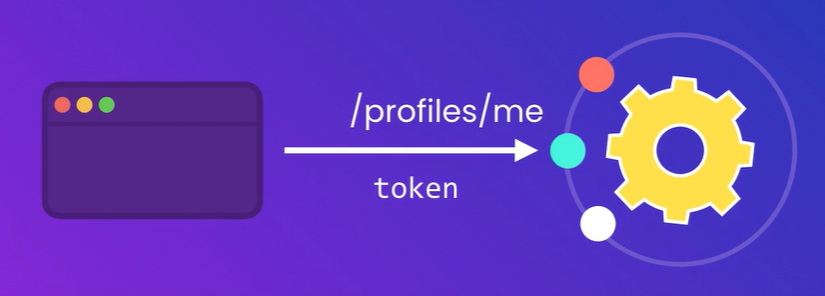
1. On the server we are going to validate the user’s credentials, *if they are not valid we are going to return an error otherwise we return a token*.



“*A token is like a temporary key we give to the client to access protected resources*”.

So the client is going to store this locally and next time it needs to access protected resources, it is going to send this token to the server.

For example, let us say a user wants to look at their profile, so the client needs to *send this request to /profiles/me and pass the token in request header*.



Now once again *on the server we will read the token, validate it and if it is valid and not expired, we will give access to this resource, otherwise we are going to return an error*.



*Valid Token* *Invalid Token*

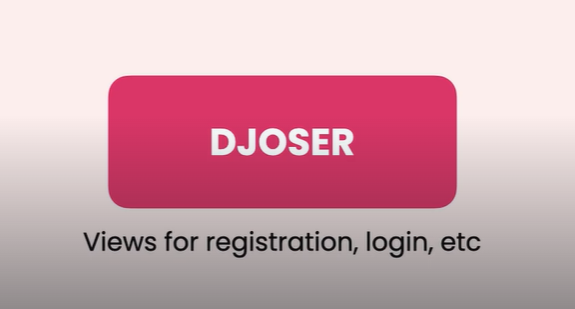
Over the next few lessons, we will see this process in action.

**Adding the Authentication Endpoints**:

So we know that Django comes with a fully fledged authentication system. But this system does not contain an API layer (*we don’t have any endpoints for users to register, login etc…*). We only have a bunch of models and database tables.



Now we can build this API layer by hand, but that is pretty tedious and repetitive, we do not want to repeat it in every project. So this is where we use a fantastic library called ***DJOSER***.



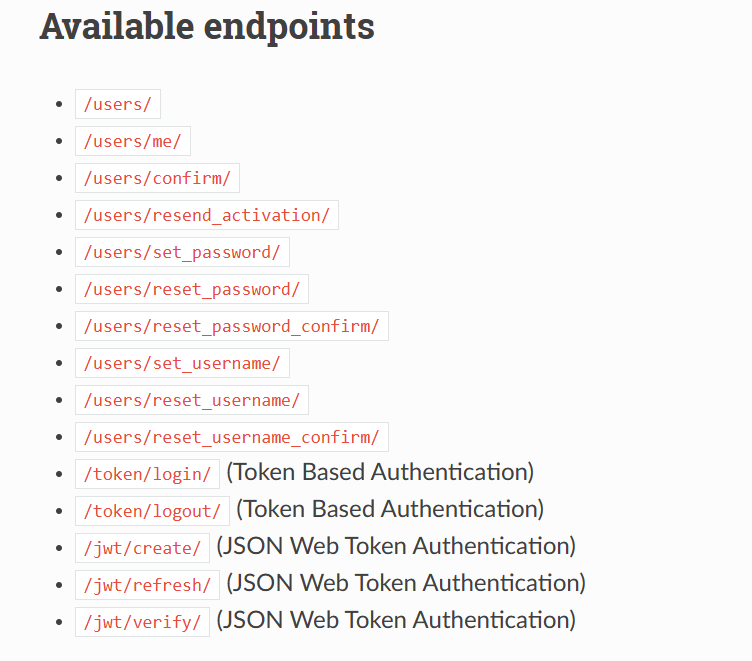
*Djoser is the RESTful implementation of Django authentication system. It provides a bunch of views for user registration, login, logout, password reset and so on*…

Go to

[Getting started — djoser 2.2.0 documentation](https://djoser.readthedocs.io/en/latest/getting_started.html)

For Djoser documentation.

Djoser provides us with a bunch of endpoints. Like /users for managing all users and /users/me for getting the current user.



*Djoser Installation*:

pipenv install djoser

*Configuration*:

Add ‘djoser’ in the list of installed apps.

    "django\_filters",

    "corsheaders",

    "rest\_framework",

    "djoser", //notice all 3rd party applications are together

*Register*:

We need to register a URL pattern in root urls.py(*storefront*). This is where we register routes from different apps.

urlpatterns = [

    path("admin/", admin.site.urls),

    path("playground/", include("playground.urls")),

    path("store/", include("store.urls")),

    path("auth/", include("djoser.urls")), 🡪 for authentication urls...

    path("\_\_debug\_\_/", include("debug\_toolbar.urls")),

]

Here we add another pattern for the authentication endpoints. We are going to delegate all these requests to *djoser.urls*.

Now *djoser relies on an authentication backend or an authentication engine to do the actual authentication*, *because djoser is just an API layer*(*a bunch of views, serializers and routes*).

So we need an authentication engine to do the actual work.

Here we have two choices, we can use

🡪 *Token based authentication* built in *Django REST framework*

**Uses a database table** to store tokens.

So *every time we receive a request on the server, to validate the token, this backend is going to go to the database to make sure this is a valid token*.

That’s going to incur a database call for every request.

OR

🡪 *JSON web token authentication* implemented in *a separate library*.

This type of authentication **does not need a database**. The way these tokens are structured, completely eliminates a database call.

*It’s because every token has a digital signature and on the server we can use that signature to ensure that this is a valid token*.

We will go with JWT *JSON web token authentication*.

*Installation*:

pipenv install djangorestframework\_simplejwt

On the Djoser’s official documentation page go to authentication backends 🡪 JSON Web Token authentication.

*Change in Django REST\_FRAMEWORK Settings*:

REST\_FRAMEWORK = {

    "COERCE\_DECIMAL\_TO\_STRING": False,

    "DEFAULT\_AUTHENTICATION\_CLASSES": ( # Added this setting here...

        "rest\_framework\_simplejwt.authentication.JWTAuthentication",

        (...),

    ),

}

Here, we are using *JWTAuthentication* class as our authentication engine.

*Add a setting*:

Add *SIMPLE\_JWT* in settings module. This step is specific to this library we just installed.

SIMPLE\_JWT = {

    "AUTH\_HEADER\_TYPES": ("JWT",),

}

With this setting we are specifying the prefix (‘*JWT*’) which should be included in the request header.

So for sending the authentication token to the server, we are going to prefix the token with ‘JWT’.

*Configure root urls.py*:

urlpatterns = [

    path("admin/", admin.site.urls),

    path("playground/", include("playground.urls")),

    path("store/", include("store.urls")),

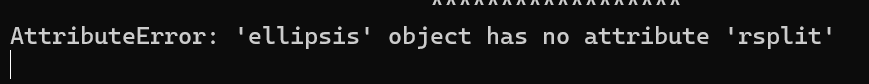
    path("auth/", include("djoser.urls")),

    path("auth/", include("djoser.urls.jwt")), 🡪 this is included for JWT

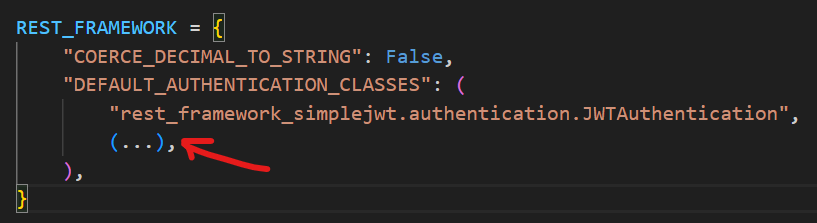
    path("\_\_debug\_\_/", include("debug\_toolbar.urls")),

]

In the terminal we have this weird error,

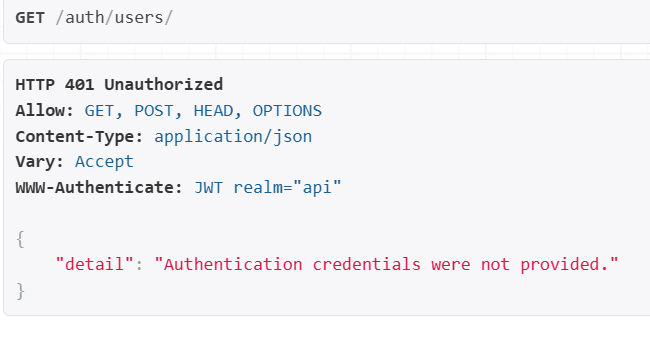


This error happened because of leaving this highlighted part in our code (*danger of copying code directly from site* 😊).



Ok back to our terminal, now we do not see any errors.

From documentation we can see that djoser provides endpoints like /users and /users/me (*but prefix them with auth/ for using them*).



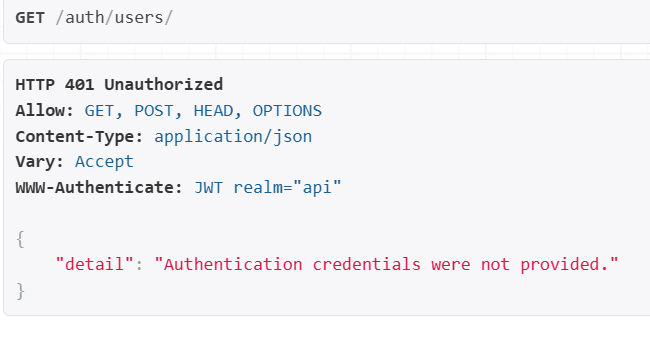
We can see this endpoint is working. But currently we cannot access it because we get a 401 unauthorized error (*since this a protected endpoint*) and not available to anonymous users.

To access this endpoint we need to pass a JSON web token in the request header, we will do that soon.

So now we have set up our authentication endpoint, next we will see how can we register new users.

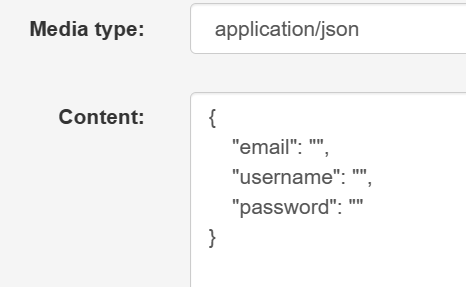
**Registering Users**:

Let us see how can we register a new user. So we have /users endpoint for managing users and as we saw in previous lesson we cannot send a GET request to this endpoint without authenticating first.



Look at the *Allow* header, here we also support POST requests and POST requests are open for anonymous users so they can register.

So to register a new user, the client app should send a POST request to this end point. Down below we have a form for registering a new user.



Let us send this object and see what happens,

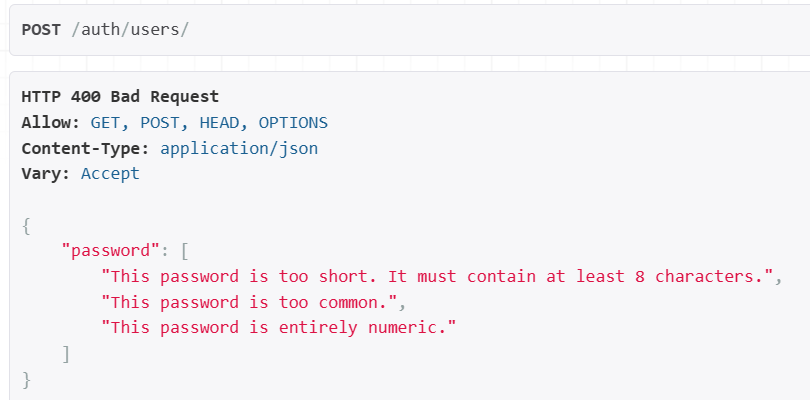
{

"email": "user1@domain.com",

"username": "user1",

"password": "1234"

}



We get a 400 bad request error because of our password being too short, common and entirely numeric.

These errors are coming from *AUTH\_PASSWORD\_VALIDATORS* in our settings module.

AUTH\_PASSWORD\_VALIDATORS = [

    {

        "NAME": "django.contrib.auth.password\_validation.UserAttributeSimilarityValidator",

    }, 🡪 This ensures password is not same as username.

    {

        "NAME": "django.contrib.auth.password\_validation.MinimumLengthValidator",

    }, 🡪 Checks the minimum length of password

    {

        "NAME": "django.contrib.auth.password\_validation.CommonPasswordValidator",

    }, 🡪 Checks if password is not a common phrase.

    {

        "NAME": "django.contrib.auth.password\_validation.NumericPasswordValidator",

    }, 🡪 Checks if password is completely numeric

]

These are various validators available to us and we can always customize them.

So let us create a new user again following these validations,

{

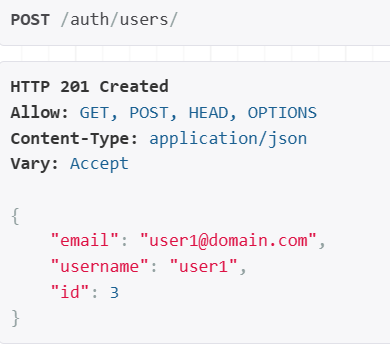
"email": "user1@domain.com",

"username": "user1",

"password": "ILoveDjango"

}

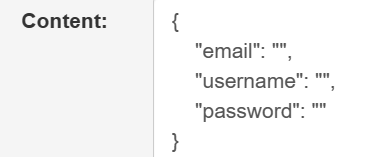
After POST we have a new user with 3 as id.



But what about first and last name?

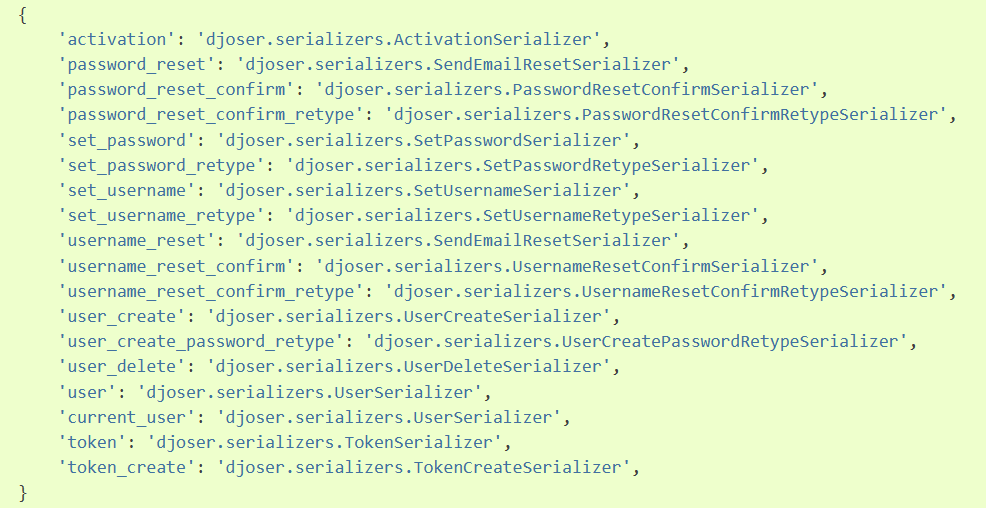
What if we want to capture them as part of the registration.

Well we know that content of this box,



Is based off of a serializer. So Djoser has a serializer for deserializing this data. So if you want to capture additional fields, we need a custom serializer.

On the official documentation page search for Settings 🡪 Serializers.



Here we see a dictionary with various serializers used in Djoser. For example for creating a new user we have a default serializer **'user\_create': 'djoser.serializers.UserCreateSerializer',**

But *using the settings module we can easily replace this with a custom serializer*.

First question comes to mind, where we are going to add this custom serializer because *we do not want to store it in our store app for the same reason that we did not want to customize the User model in this store app*.

Since this requirement is very specific to this project, how users should register on this website therefore we should add this in the *core* app.

Let us add a new file called serializers.py in our core app and import *UserCreateSerializer* class from *djoser.serializers* module.

from djoser.serializers import UserCreateSerializer

Next we need to create our own class called *UserCreateSerializer* but since we have a name clash we will give above import class an alias.

from djoser.serializers import UserCreateSerializer as BaseUserCreateSerializer

class UserCreateSerializer(BaseUserCreateSerializer):

Our custom serializer extend *BaseUserCreateSerializer*, so we get all that functionality and add something extra to it.

Let us look at the implementation of base class,

class UserCreateSerializer(UserCreateMixin, serializers.ModelSerializer):

    password = serializers.CharField(style={"input\_type": "password"}, write\_only=True)

    default\_error\_messages = {

        "cannot\_create\_user": settings.CONSTANTS.messages.CANNOT\_CREATE\_USER\_ERROR

    }

    class Meta:

        model = User

        fields = tuple(User.REQUIRED\_FIELDS) + (

            settings.LOGIN\_FIELD,

            settings.USER\_ID\_FIELD,

            "password",

        )

    def validate(self, attrs):

        user = User(\*\*attrs)

        password = attrs.get("password")

        try:

            validate\_password(password, user)

        except django\_exceptions.ValidationError as e:

            serializer\_error = serializers.as\_serializer\_error(e)

            raise serializers.ValidationError(

                {"password": serializer\_error[api\_settings.NON\_FIELD\_ERRORS\_KEY]}

            )

        return attrs

Look at the implementation of the *Meta* class. Here we have the field attribute set to an expression.

    class Meta:

        model = User

        fields = tuple(User.REQUIRED\_FIELDS) + (

            settings.LOGIN\_FIELD,

            settings.USER\_ID\_FIELD,

            "password",

        )

The result of this expression is a tuple of three values, username, password and email.

🡨These are those fields.

So back to our custom serializer, we add a Meta class here.

class UserCreateSerializer(BaseUserCreateSerializer):

    class Meta(): 🡪 This one...

But we want this Meta class to inherit everything from the Meta class in our *BaseUserCreateSerializer*.

class UserCreateSerializer(BaseUserCreateSerializer):

    class Meta(BaseUserCreateSerializer.Meta): 🡪 Meta class inherited...

Here in this custom Meta class, we only want to override the field attribute.

class UserCreateSerializer(BaseUserCreateSerializer):

    class Meta(BaseUserCreateSerializer.Meta):

        fields = ['id', 'username', 'password', 'email', 'first\_name', 'last\_name']

Here we set this fields to list (*we can also use a tuple*). We include ***id*** (*so we can return it from the server*), username, password, email, first\_name and last\_name.

Note: Technically, we could avoid extending the base Meta class and set the user attribute here directly instead of setting fields attribute. But *it is a better practice to inherit everything in the base class and override specific parts because in the next version, the implementation of the Meta class might change, so we want to inherit all that functionality and change some bits*.

So now we have a serializer, Next we need to add it in the settings module.

DJOSER = {

    'SERIALIZERS' :{}

    }

We set DJOSER to a dictionary and here we create a key called SERIALIZERS also set to a dictionary.

DJOSER = {

    'SERIALIZERS' :{

        'user\_create':

    }

    }

In this dictionary, we are going to add *user\_create*, We got that from documentation (settings 🡪 Serializers).



According to documentation, here *key is user\_create and value is the path to our custom serializer*.

DJOSER = {

    'SERIALIZERS' :{

        'user\_create':'core.serializers.UserCreateSerializer'

    }

    }

Now when we refresh the page, we see all these fields here in our auth/users endpoint.



Let us create another user with these values and POST.

{

"username": "user2",

"password": "ILoveDjango",

"email": "user2@domain.com",

"first\_name": "Joe",

"last\_name": "Smith"

}

We have a new user with these values.



Back into our serializer,

class UserCreateSerializer(BaseUserCreateSerializer):

    class Meta(BaseUserCreateSerializer.Meta):

        fields = ["id", "username", "password", "email", "first\_name", "last\_name"]

What if we want to add some profile data here as well, for example birth date. Now birth date is not part of the user table, it is part of the profile table(*Customer table*).

So here is the thing, if we add that field here directly like this,

class UserCreateSerializer(BaseUserCreateSerializer):

    class Meta(BaseUserCreateSerializer.Meta):

        fields = [

            "id",

            "username",

            "password",

            "email",

            "first\_name",

            "last\_name",

          🡪  "birth\_date",

        ]

Since this field is not a field in the user model, we have to explicitly define it here.

from djoser.serializers import UserCreateSerializer as BaseUserCreateSerializer

from rest\_framework import serializers

class UserCreateSerializer(BaseUserCreateSerializer):

    birth\_date = serializers.DateField()

    class Meta(BaseUserCreateSerializer.Meta):

        fields = [

            "id",

            "username",

            "password",

            "email",

            "first\_name",

            "last\_name",

            "birth\_date",

        ]

To save this, we need to override the save method and change how this data is saved. So first we create a user record and then we create a profile record ….

But *this is not the right way to implement this, because this serializer is purely responsible for deserializing user data and creating a user record, nothing more*.

Sure we can give it extra fields to save, but this is like a chef who also comes out of the kitchen to take orders (*doable but not the right way to manage a restaurant*).

In a proper setup every role should do one thing and do it well.



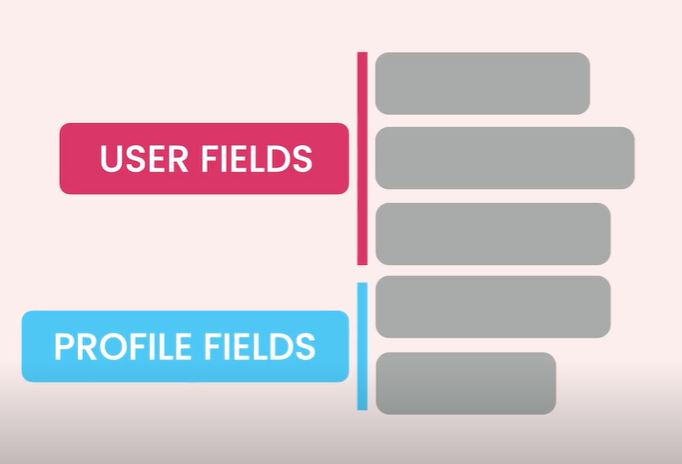
Our software should follow this philosophy.

*When we start mixing responsibility, that is when your code starts to get ugly and everything will be dependent on everything else and it will be very hard or impossible to make changes without breaking a ton of things*.

It has nothing to do with OOP or a specific framework or language, it is how we have written our code.

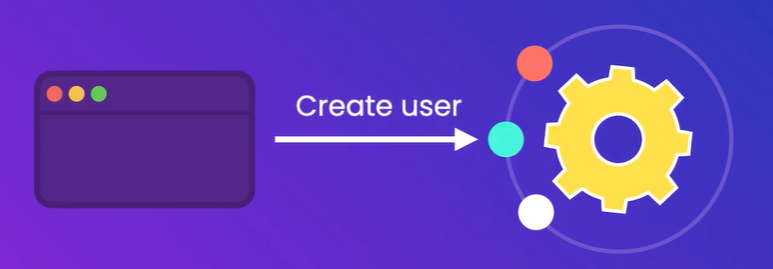
If we pay attention to these little details and properly separate responsibilities in our software end result will look beautiful and easier to maintain.

So how should we tackle this issue of adding birth\_date in our serializer. To understand it look at the client side.

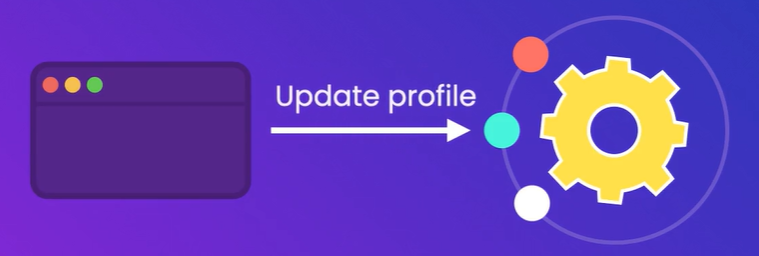


On the frontend, we are going to have a registration form and that form might include several fields. Some of these fields belong to the user model and the others belong to the profile model like the customer model.

Now to save all these fields, the client should first send a request to the users endpoint to create a user account.



Then it should send a separate request to the profile endpoint to store those additional profile related fields.



Yes, we are going to have two separate calls to the server as opposed to one call, but our endpoints are properly separated and when we properly separate these endpoints, they become more reusable and easier to maintain.

***Every endpoint should have one and only one responsibility***.

from djoser.serializers import UserCreateSerializer as BaseUserCreateSerializer

class UserCreateSerializer(BaseUserCreateSerializer):

    class Meta(BaseUserCreateSerializer.Meta):

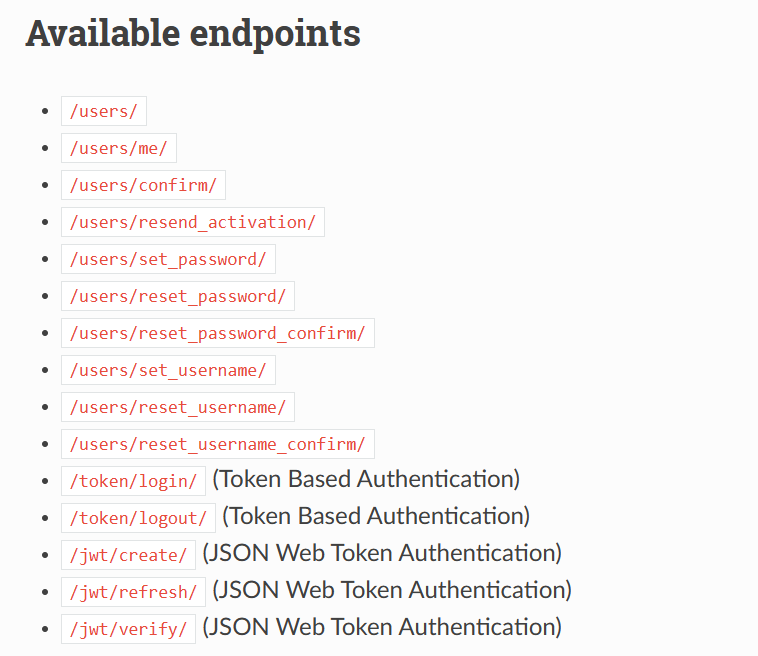
        fields = ["id", "username", "password", "email", "first\_name", "last\_name"]

So we removed the birth\_date from list of fields here and make this class simpler.

Next we will implement the profile API.

**Building the Profile API**:

Look at the endpoints provided by Djoser.



As you can see all the above endpoints are for managing users and authentication (*last ones are for authentication*). So Djoser does not have any endpoints for user profiles, which makes perfect sense (*because profiles are specific to the apps we are creating*).

So Djoser does not know we have an app called store and in that app we have a profile called customer, all its concerned about the authentication of users, nothing more.

Now we need to make customer / profile API ourselves, but where should we define it?

Well, it should be defined in store app because this is where we have defined the concept of a customer, so the API belongs to this app as well.

class CustomerSerializer(serializers.ModelSerializer):

    class Meta:

        model = Customer

        fields = []

We define a new serializer in the serializers module of the store app.

Now what fields do we need to add in our serializer, let’s take a look at the Customer model,

class Customer(models.Model):

    MEMBERSHIP\_BRONZE = "B"

    MEMBERSHIP\_SILVER = "S"

    MEMBERSHIP\_GOLD = "G"

    MEMBERSHIP\_CHOICES = [

        (MEMBERSHIP\_BRONZE, "Bronze"),

        (MEMBERSHIP\_SILVER, "Silver"),

        (MEMBERSHIP\_GOLD, "Gold"),

    ]

    phone = models.CharField(max\_length=255)

    birth\_date = models.DateField(null=True)

    membership = models.CharField(

        max\_length=1, choices=MEMBERSHIP\_CHOICES, default=MEMBERSHIP\_BRONZE

    )

    user = models.OneToOneField(settings.AUTH\_USER\_MODEL, on\_delete=models.CASCADE)

So we will add following fields,

class CustomerSerializer(serializers.ModelSerializer):

    class Meta:

        model = Customer

        fields = ["id", "user\_id", "phone", "birth\_date", "membership"]

Our serializer is ready, next we need a viewset,

class CustomerViewSet(ModelViewSet):

We can use ModelViewSet class here but its not a good choice of operations at this endpoint. For example we do not want to list customers, that is something we only need on the admin panel.

So a mobile or web app talking to our API does not really need a list of customers.

We need these operations instead,

🡪 *Create* a customer.

🡪 *Retrieve* a customer

🡪 *Update* a customer

We do not want to delete a customer, because when we delete a user, the customer record gets deleted automatically.

So here we will instead create a custom viewset using a bunch of mixins.

class CustomerViewSet(

    CreateModelMixin, RetrieveModelMixin, UpdateModelMixin, GenericViewSet

):

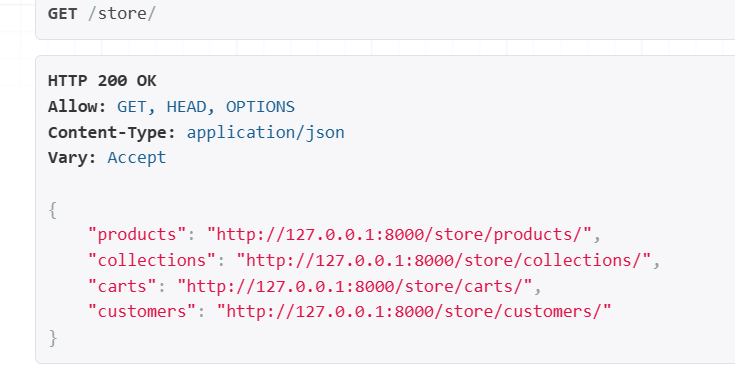
    queryset = Customer.objects.all()

    serializer\_class = CustomerSerializer

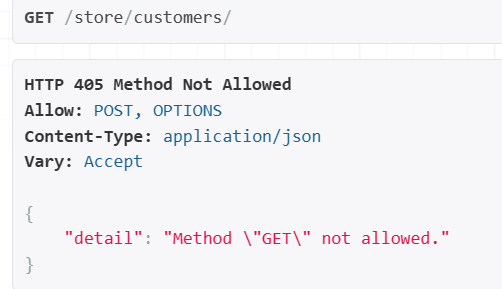
Now the last thing, we need to register the route in store app’s urls module.

router.register('customers', views.CustomerViewSet)

In the browser, if we go to store app,

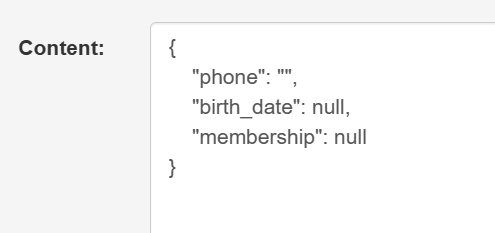


We see a new endpoint for managing our customers.



GET method is not allowed here, since we do not want client to retrieve the list of customers (*ListModelMixin not added*).

But we can create a new customer,



Here we do not have user\_id. In our *CustomerSerializer*,

class CustomerSerializer(serializers.ModelSerializer):

    class Meta:

        model = Customer

        fields = ["id", "user\_id", "phone", "birth\_date", "membership"]

Even though in the Customer model we have the user\_id attribute but this attribute is created dynamically at run time, so we need to explicitly define it here.

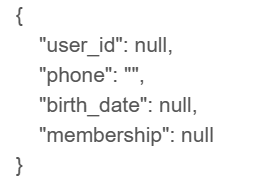
class CustomerSerializer(serializers.ModelSerializer):

    user\_id = serializers.IntegerField()

    class Meta:

        model = Customer

        fields = ["id", "user\_id", "phone", "birth\_date", "membership"]



So we see the user\_id field here now, but technically we do not need this field here because later in this section we are going to protect this API end point, so only authenticated users can all this endpoint.

Therefore *the client is going to send a token to the server and on the server we can extract the user id from the token*.

So we do not need to pass *user\_id* in the request body but for now let’s just go through with this solution because we have not implemented security yet.

Let us create a customer,

{

"user\_id": 4,

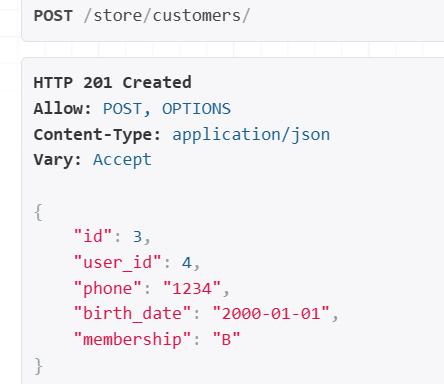
"phone": "1234",

"birth\_date": "2000-01-01",

"membership": "B"

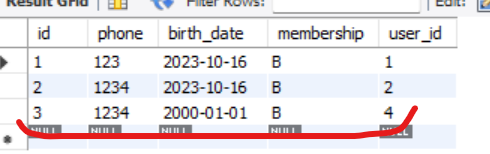
}

And POST.



Now we have a customer record for this user along with its phone, birth date and membership.

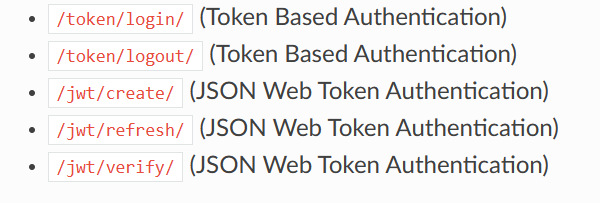
In the database, let us make sure we have this customer record in our database as well,



So our customer/ profile API is ready next we are going to talk about authenticating users.

**Logging In**:

Let us talk about authenticating users, here we have a bunch of end points for authenticating users.



Top two token based authentication comes with Django rest framework and last three endpoints are JWT authentication.

*The difference between these two authentication backends is that in token based authentication, validating a token requires a database call whereas in JWT we do not need an extra database call*.

So we will go to ***/jwt/create*** to create a new token. This is essentially the ***login end point***.

Go to ***/auth/jwt/create***.



GET method is not allowed at this endpoint but we can POST to this end point.

First let us pass an invalid password for user1.

{

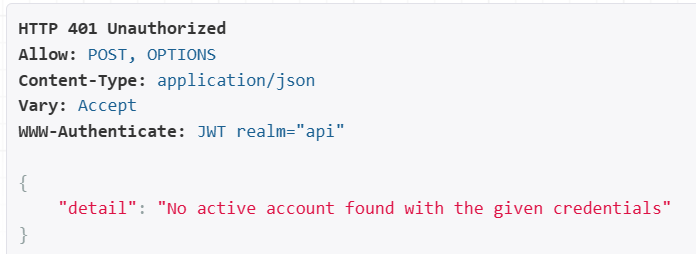
"username": "user1",

"password": "1234"

}

And POST.

We get 401 unauthorized.



Now let us login with valid credentials,

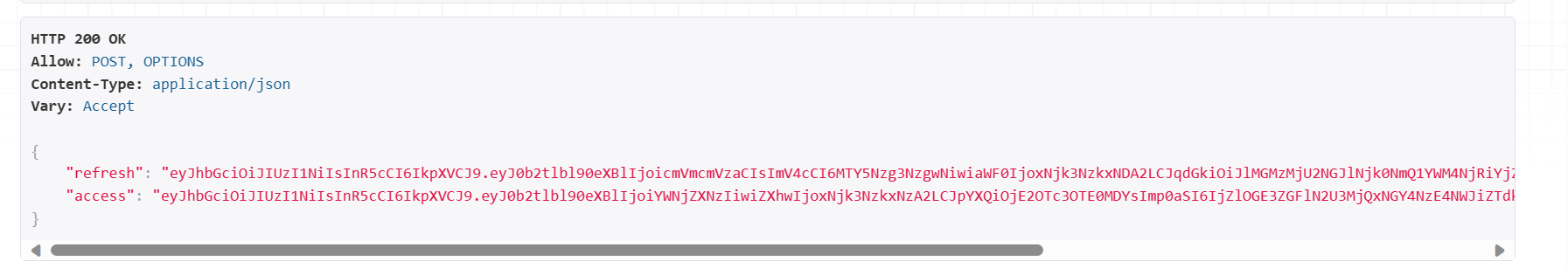
{

"username": "user1",

"password": "ILoveDjango"

}

Now we get two tokens.



We get an access token and a refresh token.

*An access token is a short lived token that we use for calling secure API end points, but when this expires we use a refresh token to get a new access token*.

Note: By default the refresh token is valid for one day, whereas the access token is valid for 5 minutes. We can easily override these settings.

If you search for Django rest framework simplejwt,

[Simple JWT — Simple JWT 5.2.2.post30+gfaf92e8 documentation (django-rest-framework-simplejwt.readthedocs.io)](https://django-rest-framework-simplejwt.readthedocs.io/en/latest/)

This was the second library we installed which is our authentication backend.

Go to settings page and we see couple of attributes,



These settings are ok but we will change it in our course. So go to settings module.

SIMPLE\_JWT = {

    "AUTH\_HEADER\_TYPES": ("JWT",),

    "ACCESS\_TOKEN\_LIFETIME": timedelta(days=1),

}

Here we changed access token lifetime to 1 day instead of 5 minutes.

So we are going to login one more time and store this access token somewhere so I can reuse it in future lessons.

"access": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ0b2tlbl90eXBlIjoiYWNjZXNzIiwiZXhwIjoxNjk3NzkxNzA2LCJpYXQiOjE2OTc3OTE0MDYsImp0aSI6IjZlOGE3ZGFlN2U3MjQxNGY4NzE4NWJiZTdkNTIwMThhIiwidXNlcl9pZCI6M30.Dt8BbIdStfRmr1GpsTALnMN7LtwMJAGICuz49o9yyOA"

Let us login again with these credentials,

{

"username": "user1",

"password": "ILoveDjango"

}

Now we get a new pair of tokens and copy the access token we get this time.

"access": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ0b2tlbl90eXBlIjoiYWNjZXNzIiwiZXhwIjoxNjk3Nzk0MTc2LCJpYXQiOjE2OTc3OTM4NzYsImp0aSI6IjQ5NmFiODI4ZmQwNTQzYTM4ZDJhOTFkZmUzZGNhNGJjIiwidXNlcl9pZCI6M30.TJASJrGNYc26HD9faluTkVSemXcyqBNfelBCob8szSo"

Let us store this in a file in our system so we can reuse it.

Note: Make sure only the access token is stored on the file no double quotes or any spaces.

**Inspecting a JSON Web Token**:

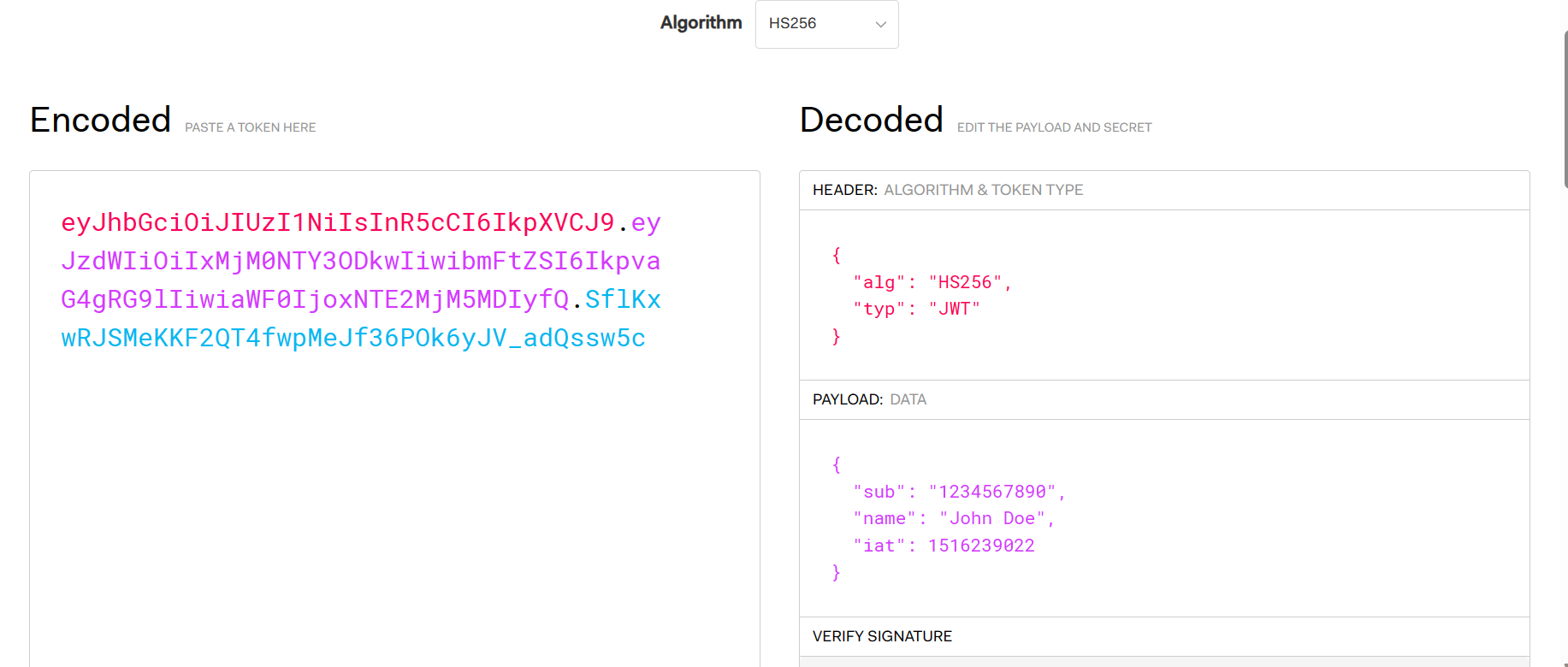
Let us demystify JSON web token. So let’s head over to jwt.io which is the official website for JSON web tokens.



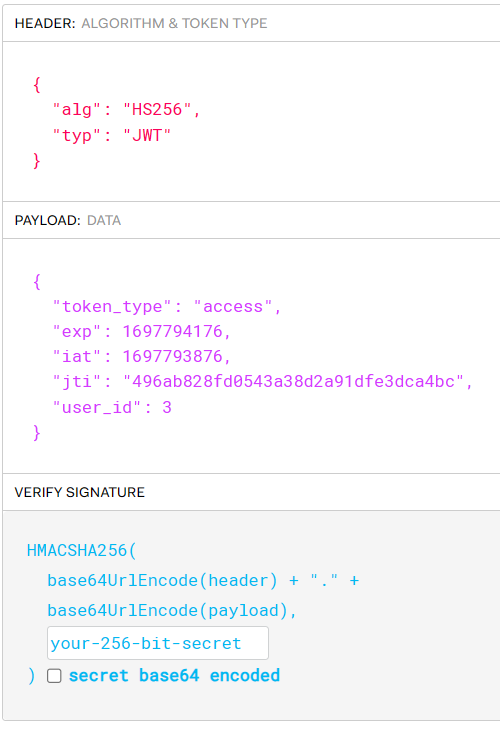
On this website we can read about them, we also have a debugger and a bunch of libraries for generating and validating JSON web tokens for different platforms.

Here *we are using* ***simple JWT****, which is for Django*. But if you are building your backend using a different platform like node or ASP.net, there are different libraries you can use to generate and validate JSON web tokens.

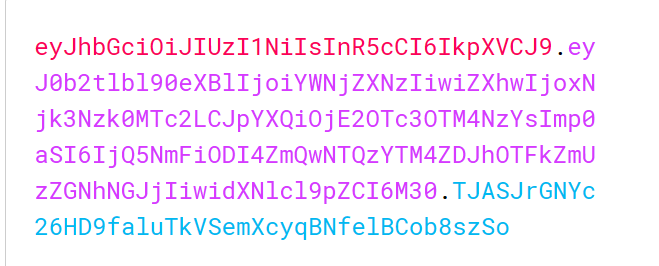
Let’s head over to debugger,



On the left side we can paste a JSON web token and see it decoded on the right side, so we will paste our JWT on the left here.



On the right side, we have a header, a payload and a signature.



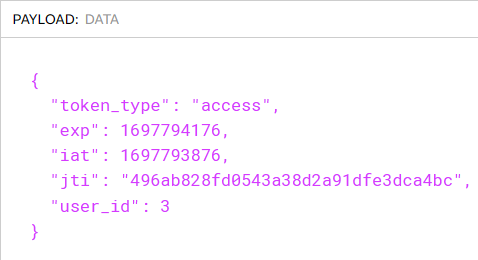
These parts are colour coded in the encoded JSON web token.

So the header is actually a JSON object that is why we called these tokens JSON web tokens.



This object has two properties called “*typ*” or type which is JWT and algorithm which is HS256.

Second part is payload.



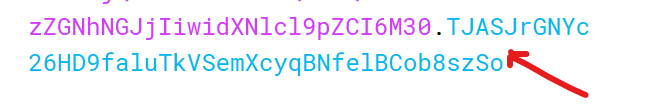
Which is again a JSON object with 4 properties, *token\_type* is access because this is an access token (*if this was refresh token we would have seen refresh here*).

Then we have expiration date and time in *exp*, *jti* a unique identifier for this token and finally our *user\_id*.

*Where did this user\_id come from*?

*When we login our authentication backend or our authentication engine validates our credentials. If they are valid, it is going to retrieve our user account and then it will grab our user\_id and puts it in our JSON web token*.

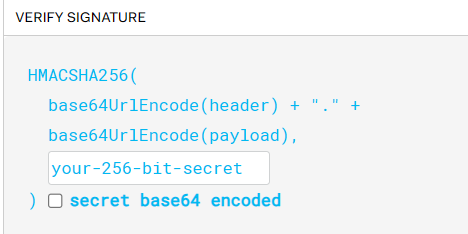
Now the most important part of this token, is the digital signature in blue.



This digital signature is generated based off of the header and the payload. So if you make any changes in the payload, the signature has to be regenerated.

So if a hacker grabs your JSON web token somehow and then brings it to this debugger and tries to change the user id to someone else’s user id then the signature has to be regenerated. If they don’t regenerate the signature and send this token to the server, server is going to say Hey, this is an invalid token, we did not generate this because this signature does not match our signature (*it is like a fake driver’s license*).

You might be wondering if a hacker can regenerate the signature. In theory, YES but that’s only possible if hacker gets access to the server. That is where we have secret for generating digital signatures.



Over here we can see the formula for generating the signature.

We grab the header + Payload and then using a 256 bit secret, we generate a signature.

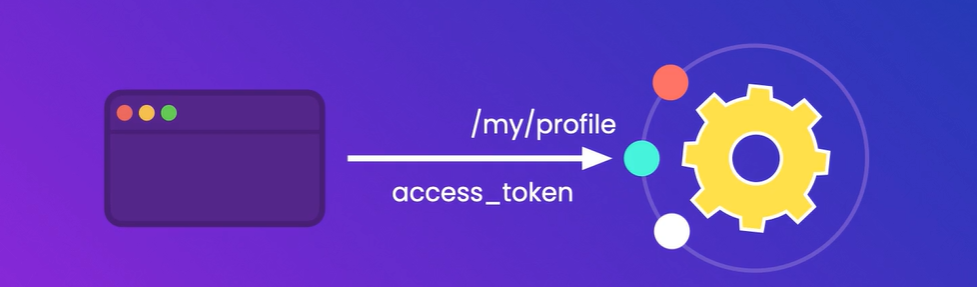
Now this 256 bit secret is only stored on the server. So unless the hacker gets access to our server they cannot regenerate this digital signature and whatever changes they apply to the payload essentially becomes invalid.

*So this is how JSON web tokens work*.

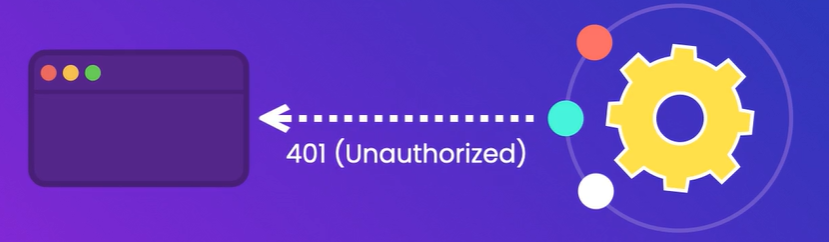
**Refreshing Tokens**:

Let us do a quick recap, When we login, we get two tokens, an access token that is valid for 5 minutes by default and a refresh token that is valid for one day by default.

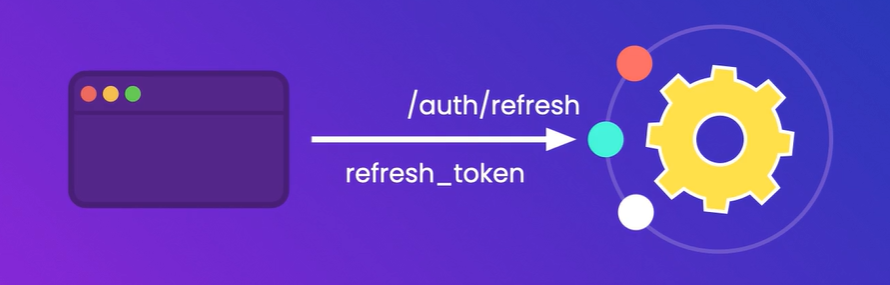
Now if the client wants to access a protected API endpoint, it needs to send the access token to the server in the request header.

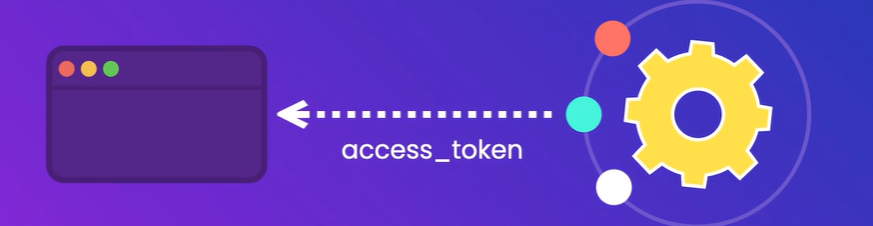


But if the token is expired, server is going to respond with a 401 error meaning unauthorized.



At this point, the *client needs to call the refresh endpoint using the refresh token to get a new access token*.





Once it gets the new access token then it can call protected endpoints.

So in the list of available endpoints, we also have /***jwt/refresh/***.

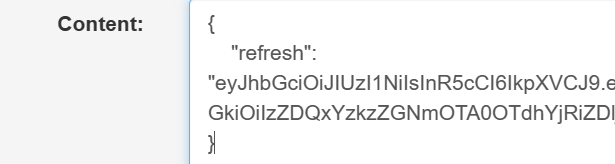
for refreshing a JSON web token.

So let us see how this works,

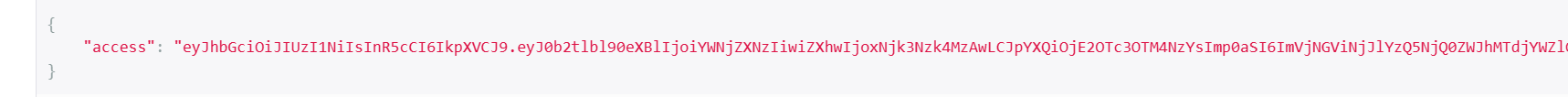
"refresh": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ0b2tlbl90eXBlIjoicmVmcmVzaCIsImV4cCI6MTY5Nzg4MDI3NiwiaWF0IjoxNjk3NzkzODc2LCJqdGkiOiIzZDQxYzkzZGNmOTA0OTdhYjRiZDljNDI1ZWY4NWI0OCIsInVzZXJfaWQiOjN9.B8nInkKbURleaHpPUJnsEoY1ru38vqefSaIjXBmeiP8",

This is my current refresh token.

We will go to /***jwt/refresh/*** end point and paste our refresh token in body of POST request.



And now we get a new access token in response.



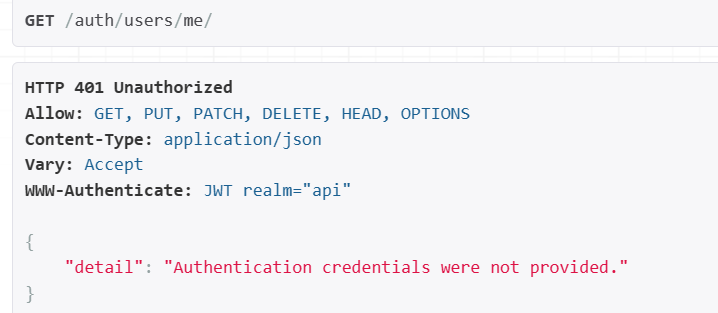
"access": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ0b2tlbl90eXBlIjoiYWNjZXNzIiwiZXhwIjoxNjk3Nzk4MzAwLCJpYXQiOjE2OTc3OTM4NzYsImp0aSI6ImVjNGViNjJlYzQ5NjQ0ZWJhMTdjYWZlODQyOGRlM2Q2IiwidXNlcl9pZCI6M30.o8p4EWw8eZ-xBognevgwzshnNQzdQUqr87Nw1syJV8c"

}

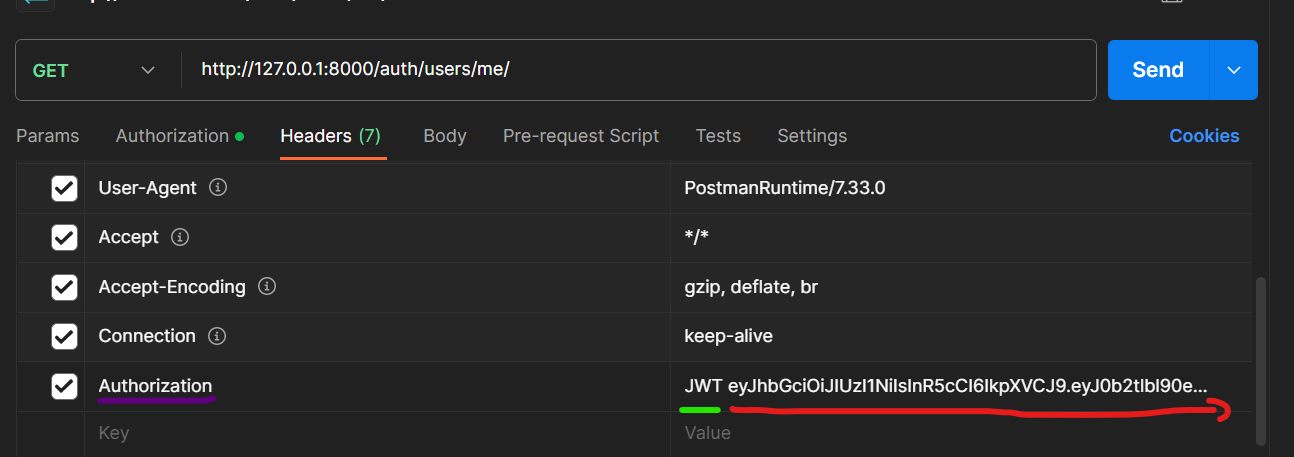
**Getting the Current User**:

We learned how to allow users to register and login, now let us see how we can get the current user. We have an end point for that called ***/users/me***.

If we hit this endpoint,



We get 401 unauthorized error, because we have not supplied our authentication credentials. So here we need to pass our access token to call this endpoint.



I used postman here to modify request header. Here “Authorization” is the name of the request header and we set its value to JWT followed by our access token.

The reason we used JWT in the Authorization value is because of our *SIMPLE\_JWT* setting (*JWT is the prefix for setting authorization header types*).

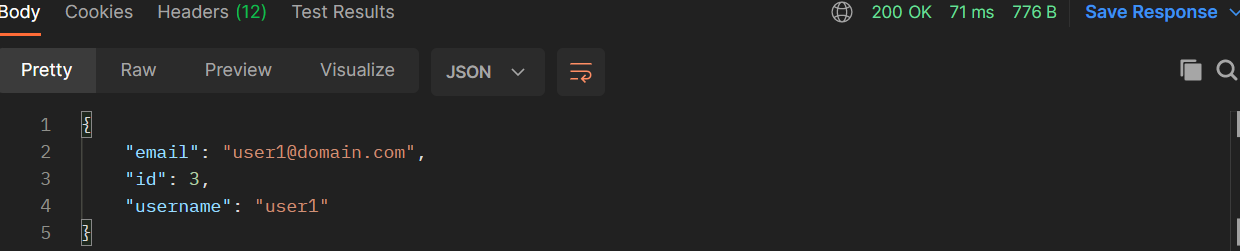
SIMPLE\_JWT = {

    "AUTH\_HEADER\_TYPES": ("JWT",), 🡪 *This one...*

    "ACCESS\_TOKEN\_LIFETIME": timedelta(days=1),

}

Now we send this request we get below response,



But what if we want to include first and last names as well, for that we need a custom serializer.

In Djoser documentation, search for serializers,



We have a serializer for getting the current user, so to create first and last names, we just need to modify this serializer.

Back to serializers module of the core app, let us import this *UserSerializer*.

from djoser.serializers import (

    UserSerializer as BaseUserSerializer, 🡪imported with an alias

    UserCreateSerializer as BaseUserCreateSerializer,

)

Then we need to create our own custom class where we redefine our fields attribute,

class UserSerializer(BaseUserSerializer):

    class Meta(BaseUserSerializer.Meta):

        fields = ["id", "email", "username", "first\_name", "last\_name"]

Finally we need to register this in our settings module,

DJOSER = {

    "SERIALIZERS": {

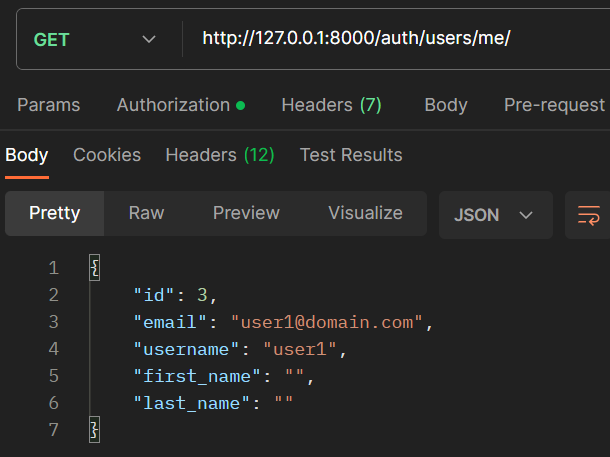
        "user\_create": "core.serializers.UserCreateSerializer",

        "current\_user": "core.serializers.UserSerializer", 🡪added here...

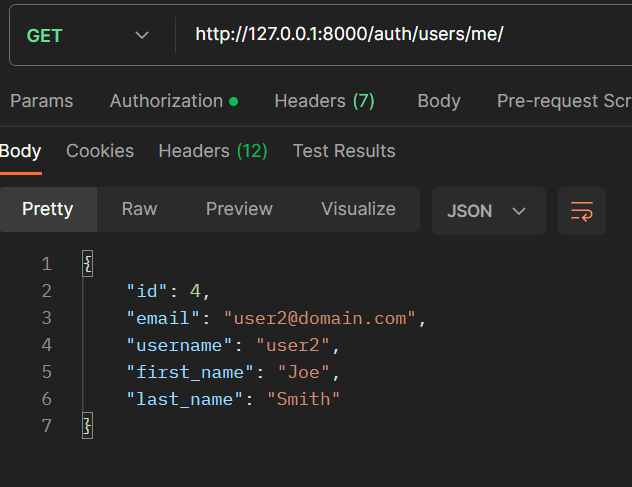
    }

}

We see our newly added fields in response now,



Here first and last names are empty because we did not supply them while registering user1. So if we try to login as user2 and get a new access token, we will see first and last names of user which is “Joe Smith”.



Next we will talk about getting the current’s user profile.

**Getting Current User’s Profile**:

Alright now let’s see how we can get or update current user profile.

So we are going to add a new endpoint like this,

<http://127.0.0.1:8000/store/customers/me>

Back to our customer viewset, we want to define a new method called *me*() with two parameters self and request.

class CustomerViewSet(

    CreateModelMixin, RetrieveModelMixin, UpdateModelMixin, GenericViewSet

):

    queryset = Customer.objects.all()

    serializer\_class = CustomerSerializer

    def me(self, request): 🡪 an action...

Now this method is more accurately called an *action*. So *all the methods we have here in this viewset responding to request are called action*.

We have *create* method we got from CreateModelMixin, we also have *retrieve* method that we inherited from

RetrieveModelMixin and so on. All these methods are called actions.

In this case *we are defining a custom action so we need to decorate it with the action decorator in rest framework*.

from rest\_framework.decorators import action

*Setting the detail argument on action decorator to* ***False***,

    @action(detail=False)

    def me(self, request):

if we set detail to False, that means this action is available on the list view.

It will be available here, <http://127.0.0.1:8000/store/customers/me>

It is because customers is a list.

*Setting the detail argument on action decorator to* ***True***,

If we set detail to True, action is going to be available on the detail view e.g. this URL.

<http://127.0.0.1:8000/store/customers/1/me>

It means we will go to a specific customer and then access this end point.

In this case we want the action to be available on the list, so we set detail to false.

class CustomerViewSet(

    CreateModelMixin, RetrieveModelMixin, UpdateModelMixin, GenericViewSet

):

    queryset = Customer.objects.all()

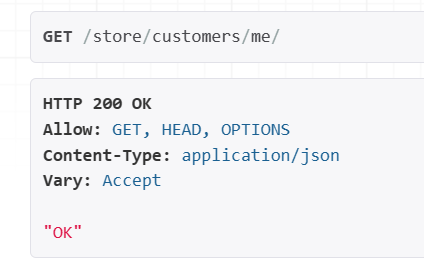
    serializer\_class = CustomerSerializer

    @action(detail=False)

    def me(self, request):

        return Response("OK")

Here we just return a simple Response of “OK” to make sure our plumbing is working fine.



Now next step is to retrieve a customer and return it to client.

We saw that our JWT contains a user\_id in its payload. Let us see how can we get that?

Well earlier we talked about *AuthenticationMiddleware*.

MIDDLEWARE = [

    "corsheaders.middleware.CorsMiddleware",

    "debug\_toolbar.middleware.DebugToolbarMiddleware",

    "django.middleware.security.SecurityMiddleware",

    "django.contrib.sessions.middleware.SessionMiddleware",

    "django.middleware.common.CommonMiddleware",

    "django.middleware.csrf.CsrfViewMiddleware",

    "django.contrib.auth.middleware.AuthenticationMiddleware", 🡪 this one...

    "django.contrib.messages.middleware.MessageMiddleware",

    "django.middleware.clickjacking.XFrameOptionsMiddleware",

]

The job of this middleware is *to inspect the incoming request and if there is information about the user, its going to retrieve that user from the database and attach it to the request object*.

Back in our CustomerViewset, every *request* has a *user* attribute and *if the user is not logged in, this is going to be set to an instance of anonymous user class otherwise this is going to be an user object*.

class CustomerViewSet(

    CreateModelMixin, RetrieveModelMixin, UpdateModelMixin, GenericViewSet

):

    queryset = Customer.objects.all()

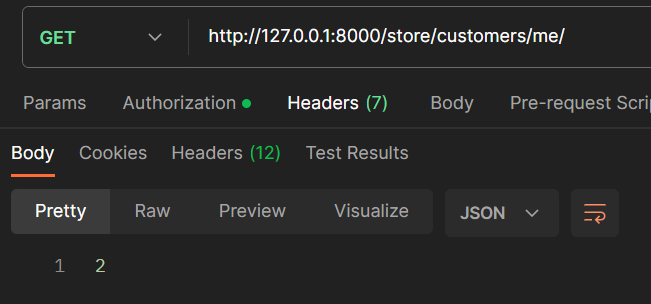
    serializer\_class = CustomerSerializer

    @action(detail=False)

    def me(self, request):

        return Response(request.user.id)

Instead of returning “OK” , we return *request.user.id*.

🡨We get 2(this request should pass here with authorization token then we get id).

Since we get user\_id in response now, we can easily retrieve customer information with this user ID and return the response to client.

class CustomerViewSet(

    CreateModelMixin, RetrieveModelMixin, UpdateModelMixin, GenericViewSet

):

    queryset = Customer.objects.all()

    serializer\_class = CustomerSerializer

    @action(detail=False)

    def me(self, request):

        customer = Customer.objects.get(user\_id=request.user.id)

        serializer = CustomerSerializer(customer)

        return Response(serializer.data)

Here is my profile / customer record.



Currently we cannot update this profile, as you can see in allow header we can only send these requests.



So now we need to enable PUT requests.

Back to our action, we need to set another argument called *methods*. We want to support GET and PUT based on the request method.

    @action(detail=False, methods=["GET", "PUT"]) 🡪 add here...

    def me(self, request):

Then we implement “PUT” request handing,

class CustomerViewSet(

    CreateModelMixin, RetrieveModelMixin, UpdateModelMixin, GenericViewSet

):

    queryset = Customer.objects.all()

    serializer\_class = CustomerSerializer

    @action(detail=False, methods=["GET", "PUT"])

    def me(self, request):

        (customer, created) = Customer.objects.get\_or\_create(user\_id=request.user.id)🡪 *why get\_or\_create???*

        if request.method == "GET": 🡪 For getting customer info

            serializer = CustomerSerializer(customer)

            return Response(serializer.data)

        elif request.method == "PUT": 🡪 For updating customer info

            serializer = CustomerSerializer(customer, data=request.data)

            serializer.is\_valid(raise\_exception=True)

            serializer.save()

            return Response(serializer.data)

(***Note****, I could use DRF browsable API page only because I hardcoded user\_id to 2 because I cannot send authorization token in browser, only possible in POSTMAN*)

But look we don’t want to update user\_id field here because we do not want to associate this profile with someone else’s account.

So *user\_id* must be read only.

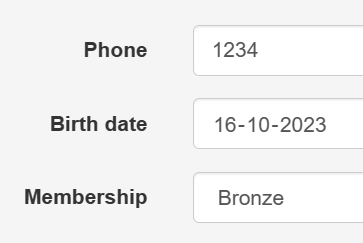
class CustomerSerializer(serializers.ModelSerializer):

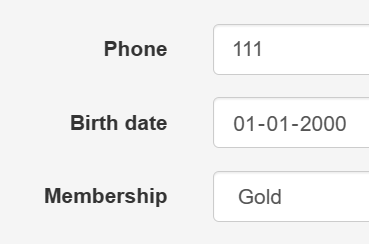
    user\_id = serializers.IntegerField(read\_only=True) 🡪 change here...

    class Meta:

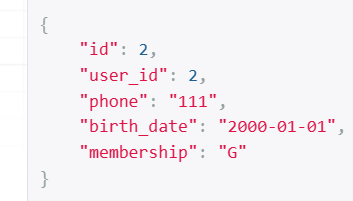
        model = Customer

        fields = ["id", "user\_id", "phone", "birth\_date", "membership"]

, now we will update the fields here.



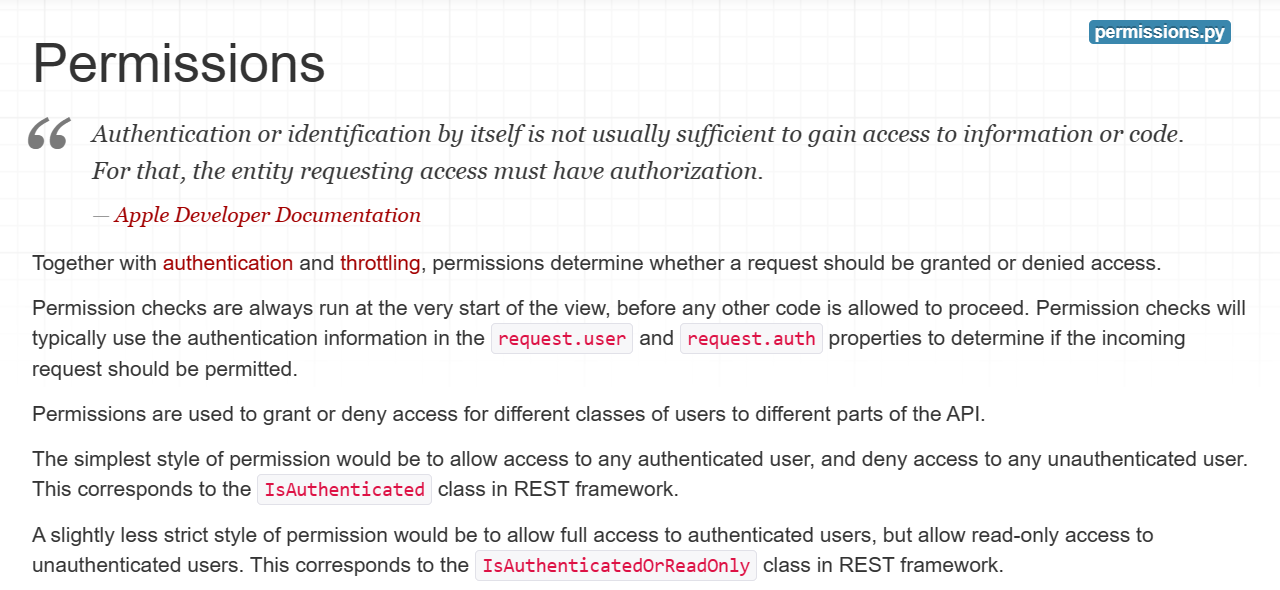
Now our customer is updated,



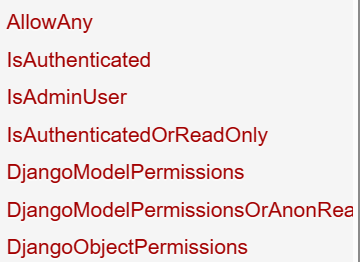
Now we can get or update current users profile, but we need to apply a permission to this endpoint, so it is only accessible by authenticated users.

**Applying Permissions**:

On the Django-rest-framework.org, go to API Guide then permissions.



Here we have a bunch of permission classes like *AllowAny* which is applied globally by default (*this is why we can access all API endpoints without logging in*).



We also have *IsAuthenticated*, *IsAdminUser*, *IsAuthenticatedOrReadOnly* and we can also create our own custom permission class.

For now let us see how can we use these permission classes. One way is to apply them globally.

So here in the settings module, inside settings for the REST\_FRAMEWORK, just like we have *DEFAULT\_AUTHENTICATION\_CLASSES*, we can also add ***DEFAULT\_PERMISSION\_CLASSES***.

    "DEFAULT\_PERMISSION\_CLASSES":[

        'rest\_framework.permissions.AllowAny' 🡪 *This is the default value*

    ]

We can change it to *IsAuthenticated*.

REST\_FRAMEWORK = {

    "COERCE\_DECIMAL\_TO\_STRING": False,

    "DEFAULT\_AUTHENTICATION\_CLASSES": (

        "rest\_framework\_simplejwt.authentication.JWTAuthentication",

    ),

    "DEFAULT\_PERMISSION\_CLASSES":[

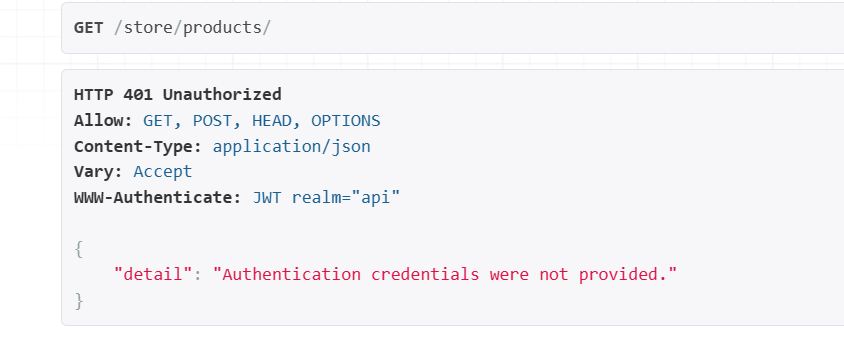
        'rest\_framework.permissions.IsAuthenticated' *🡪 change Here ...*

    ]

}

Now *all of our API end points are closed to Anonymous users*.

If you go to incognito tab and try to see list of products,



Now we have to authenticate in order to see the list of products. So *we can apply one or more permission classes globally and then if we want to we can always override these permissions for specific views*.

Let us delete this global permission and apply some permissions in our viewset.

class CustomerViewSet(

    CreateModelMixin, RetrieveModelMixin, UpdateModelMixin, GenericViewSet

):

    queryset = Customer.objects.all()

    serializer\_class = CustomerSerializer

    permission\_classes = [] 🡪 Here...

Here we can set *permission\_classes* to a list of one or more permissions. *We use list here because we can supply multiple permissions here and if any of them fails then the client will not be able to access this view*.

First lets import *IsAuthenticated* class from rest\_framework.

from rest\_framework.permissions import IsAuthenticated

Then we add this class in our ViewSet,

class CustomerViewSet(

    CreateModelMixin, RetrieveModelMixin, UpdateModelMixin, GenericViewSet

):

    queryset = Customer.objects.all()

    serializer\_class = CustomerSerializer

    permission\_classes = [IsAuthenticated] 🡪 Here...

    @action(detail=False, methods=["GET", "PUT"])

    def me(self, request):

        (customer, created) = Customer.objects.get\_or\_create(user\_id=request.user.id

)

        if request.method == "GET":

            serializer = CustomerSerializer(customer)

            return Response(serializer.data)

        elif request.method == "PUT":

            serializer = CustomerSerializer(customer, data=request.data)

            serializer.is\_valid(raise\_exception=True)

            serializer.save()

            return Response(serializer.data)

Now all actions in this viewset are closed for anonymous users.

But then again if we want to we can override this permission in a particular action.

@action(detail=False, methods=["GET", "PUT"], permission\_classes=[]) 🡪 *like this*

What if we want to have different permissions for different actions like create and retrieve. For example *we want anyone to be able to retrieve a customer object but only authenticated users or admin users can update a customer object*.

For that we need to override a method available in this viewset called *get\_permissions*.

    def get\_permissions(self):

        if self.request.method == "GET":

            return [AllowAny()] 🡪 *List of objects not classes...*

Here we checked to see if *self.request.method* == “GET” then we can return a list of AllowAny ‘***objects’***.

Note: We return list of objects from *get\_permissions* method as opposed to list of classes in *permission\_classes* attribute. If we do not add (*parentheses*) we will get weird exception at runtime

class CustomerViewSet(

    CreateModelMixin, RetrieveModelMixin, UpdateModelMixin, GenericViewSet

):

    queryset = Customer.objects.all()

    serializer\_class = CustomerSerializer

    permission\_classes = [IsAuthenticated]

    def get\_permissions(self): -🡪 *Permission override for request*

        if self.request.method == "GET":

            return [AllowAny()] 🡪 *no authentication required*

        return [IsAuthenticated()] 🡪 *authentication required*

    @action(detail=False, methods=["GET", "PUT"])

    def me(self, request):

        (customer, created) = Customer.objects.get\_or\_create(user\_id=request.user.id

)

        if request.method == "GET":

            serializer = CustomerSerializer(customer)

            return Response(serializer.data)

        elif request.method == "PUT":

            serializer = CustomerSerializer(customer, data=request.data)

            serializer.is\_valid(raise\_exception=True)

            serializer.save()

            return Response(serializer.data)

So here we allow unrestricted access to GET method but for anything else we want the user to be authenticated.

We can GET customer

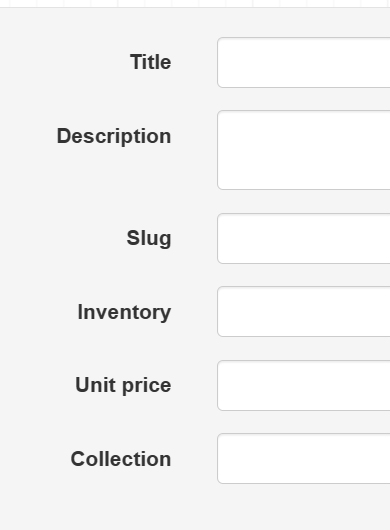


But there is no PUT option below.

So this is how permissions work in DRF.

**Applying Custom Permissions**:

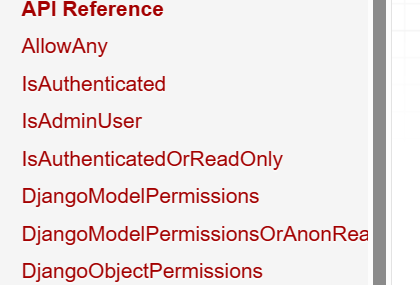
This is our products endpoint.

Currently this endpoint is open for everyone including anonymous users. We can create a new product and even update or delete products.

Certainly we do not want this to happen, *we only want admin users to be able to modify products but anyone including anonymous users should be able to retrieve the list of products*.

On DRF permissions API reference page, we currently do not have any class called *IsAdminOrReadOnly* we only have *IsAutheticatedOrReadOnly*.



This is a situation where we need to create a custom permission and that is super easy.

Back in the store app, create a new file called permissions.py. On the top from rest\_framework.*permissions* we will import *IsAuthenticated* and *BasePermission* classes.

from rest\_framework.permissions import IsAuthenticated, BasePermission

Let us look at the implementation of *IsAuthenticated* class,

class IsAuthenticated(BasePermission):

    """

    Allows access only to authenticated users.

    """

    def has\_permission(self, request, view):

        return bool(request.user and request.user.is\_authenticated)

This class extends *BasePermission* (*which is the base class for all permissions in DRF*).

Here we have a single method called *has\_permission* which returns a Boolean value. Inside it there is a simple condition if *request.user* is set AND this user is authenticated then we return True.

Earlier we saw that *if the user is not logged in then request.user is set to an instance of anonymous user class and in that class if we read is\_authenticated we get false*. This is the implementation of IsAuthenticated class.

Let us look at another class called *IsAdminUser*.

class IsAdminUser(BasePermission):

    """

    Allows access only to admin users.

    """

    def has\_permission(self, request, view):

        return bool(request.user and request.user.is\_staff)

Here we are saying if request.user is set and request.user is a staff member (*is\_staff = True, meaning it can log into admin panel*) then we are going to return True.

So now we understand, how permissions work, so let us create our own custom permission class.

We need only BasePermission class for this purpose,

from rest\_framework.permissions import BasePermission

We will create a class called *IsAdminOrReadOnly* which extends *BasePermission*.

class IsAdminOrReadOnly(BasePermission):

    def has\_permission(self, request, view):

        return

We are going to override *has\_permission* method.

Next we check if *request.method* == ‘GET’ then we return True which means anyone can access the target view.

class IsAdminOrReadOnly(BasePermission):

    def has\_permission(self, request, view):

        if request.method == "GET":

            return True

For all the other methods, we check if request.user is set and is a staff member (*an admin user*).

from rest\_framework.permissions import BasePermission

class IsAdminOrReadOnly(BasePermission):

    def has\_permission(self, request, view):

        if request.method == "GET":

            return True

        return bool(request.user and request.user.is\_staff) 🡪 *use bool*

This is how we can create a custom permission class.

But there is a tiny issue here and that is with the current implementation even HEAD or OPTIONS request require user to be an admin (*we certainly do not want this to be the case*).



So the right way to write our condition is to *see if our method is in list of safe request methods*.

from rest\_framework import permissions

From rest framework we import *permissions* module which has a constant called *SAFE\_METHODS*. Let us look at its implementation.

SAFE\_METHODS = ('GET', 'HEAD', 'OPTIONS')

In our custom permission class,

from rest\_framework.permissions import BasePermission

from rest\_framework import permissions

class IsAdminOrReadOnly(BasePermission):

    def has\_permission(self, request, view):

        if request.method in permissions.SAFE\_METHODS:

            return True

        return bool(request.user and request.user.is\_staff)

If our request.method is in the list of safe methods return True otherwise we have another condition specified.

We can consolidate two import statements in our class like this,

from rest\_framework import permissions

class IsAdminOrReadOnly(permissions.BasePermission):

    def has\_permission(self, request, view):

        if request.method in permissions.SAFE\_METHODS:

            return True

        return bool(request.user and request.user.is\_staff)

Let us apply our custom permission to product viewset.

class ProductViewSet(ModelViewSet):

    queryset = Product.objects.all()

    serializer\_class = ProductSerializer

    filter\_backends = [DjangoFilterBackend, SearchFilter, OrderingFilter]

    filterset\_class = ProductFilter

    pagination\_class = DefaultPagination

    permission\_classes= [IsAdminOrReadOnly] 🡪 *Here*

    search\_fields = ["title", "description"]

    ordering\_fields = ["unit\_price", "last\_update"]

    def get\_serializer\_context(self):

        return {"request": self.request}

    def destroy(self, request, \*args, \*\*kwargs):

        if OrderItem.objects.filter(product\_id=kwargs["pk"]).count() > 0:

            return Response(

                {

                    "error": "This product cannot be deleted because it is associated with an order item"

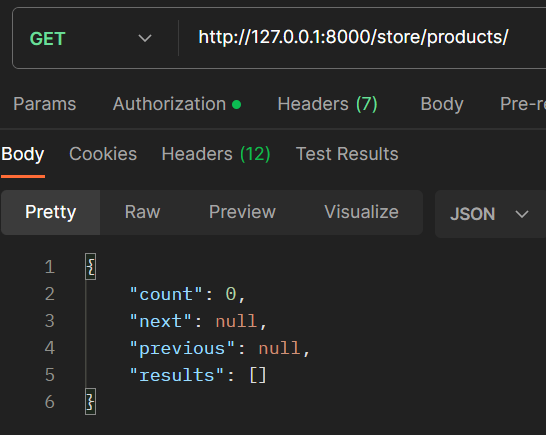
                },

                status=status.HTTP\_405\_METHOD\_NOT\_ALLOWED,

            )

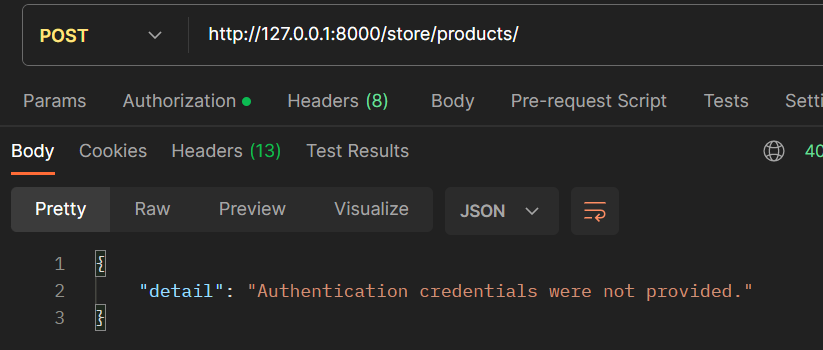
        return super().destroy(request, \*args, \*\*kwargs)

Now Let us test our implementation,

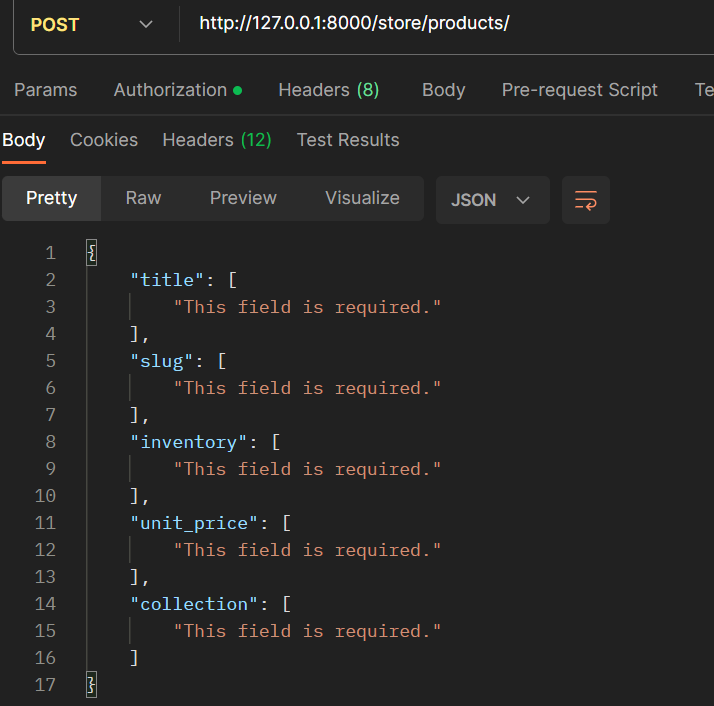


I can GET the list of products without access token.

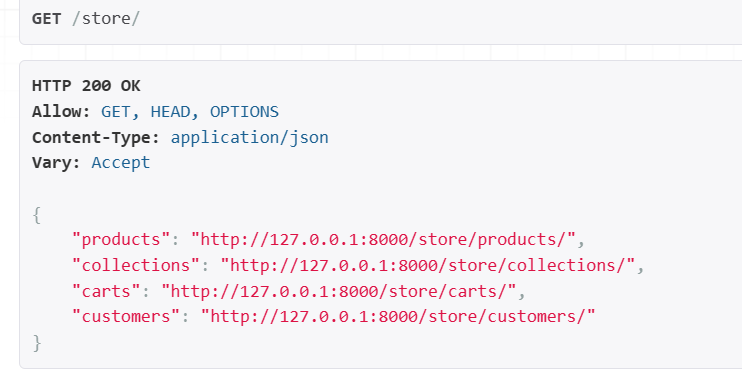
But when I try to create a product using POST without access token,



In contrast same POST request with access token,



Now let us apply the same security to other API end points,



First Collections end point,

class CollectionViewSet(ModelViewSet):

    queryset = Collection.objects.annotate(products\_count=Count("product")).all()

    serializer\_class = CollectionSerializer

    permission\_classes = [IsAdminOrReadOnly]

    def destroy(self, request, \*args, \*\*kwargs):

        if Product.objects.filter(collection\_id=kwargs["pk"]).count() > 0:

            return Response(

                {

                    "error": "This collection cannot be deleted because it is associated with a product"

                },

                status=status.HTTP\_405\_METHOD\_NOT\_ALLOWED,

            )

        return super().destroy(request, \*args, \*\*kwargs)

For carts end point, we do not want to apply any permission here because anyone including anonymous users should be able to create a shopping cart.

For Customers end point, we want to make sure only authenticated users can access this end point.

class CustomerViewSet(

    CreateModelMixin, RetrieveModelMixin, UpdateModelMixin, GenericViewSet

):

    queryset = Customer.objects.all()

    serializer\_class = CustomerSerializer

    permission\_classes = [IsAuthenticated] 🡪 *Permission applied here*

    def get\_permissions(self):

        if self.request.method == "GET":

            return [AllowAny()]

        return [IsAuthenticated()]

    @action(detail=False, methods=["GET", "PUT"])

    def me(self, request):

        (customer, created) = Customer.objects.get\_or\_create(user\_id=request.user.id

)

        if request.method == "GET":

            serializer = CustomerSerializer(customer)

            return Response(serializer.data)

        elif request.method == "PUT":

            serializer = CustomerSerializer(customer, data=request.data)

            serializer.is\_valid(raise\_exception=True)

            serializer.save()

            return Response(serializer.data)

There is a problem with this implementation. Earlier we used these mixins here CreateModelMixin, RetrieveModelMixin and UpdateModelMixin because we assumed that using this API we do not want to list or delete our customers.

We assumed that we want to perform these tasks via the admin panel. But let us change the requirements a little bit. Let’s make this endpoint consistent with our Products and Collections end points.

So we want to allow all operations but we want to restrict them only to admin users (*only admins can manage customers but any authenticated user must be able to access their profile*).

Therefore instead of all these individual mixins, let’s use ModelViewSet as the base class (*so all operations are available*) then we change the permissions class to *IsAdminUser* (*defined in REST framework*) and then we will override this permission for our *me* action.

class CustomerViewSet(ModelViewSet):

    queryset = Customer.objects.all()

    serializer\_class = CustomerSerializer

    permission\_classes = [IsAdminUser] 🡪 *permission defined from DRF*

    def get\_permissions(self):

        if self.request.method == "GET": 🡪 Is this part still required?

            return [AllowAny()] NOT SURE, will comment out

        return [IsAuthenticated()]

    @action(detail=False, methods=["GET", "PUT"], permission\_classes=[IsAuthenticated]) 🡪 *override permission for custom action*

    def me(self, request):

        (customer, created) = Customer.objects.get\_or\_create(user\_id=request.user.id

)

        if request.method == "GET":

            serializer = CustomerSerializer(customer)

            return Response(serializer.data)

        elif request.method == "PUT":

            serializer = CustomerSerializer(customer, data=request.data)

            serializer.is\_valid(raise\_exception=True)

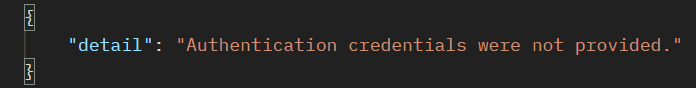
            serializer.save()

            return Response(serializer.data)

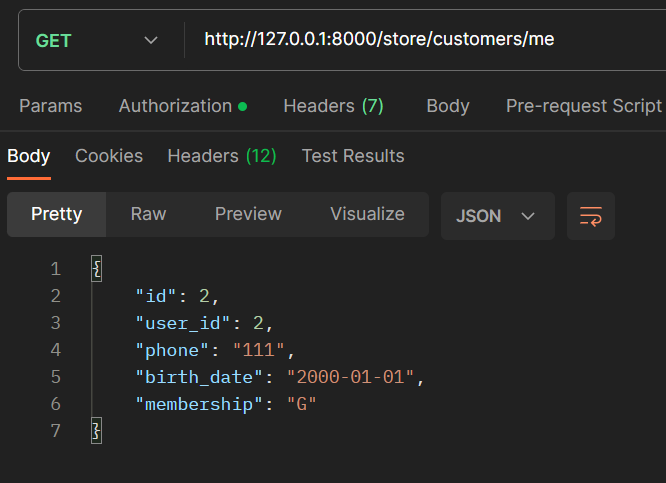
Now if we have access token with a staff member(*admin*), we can get the list of customers,



But without access token,



Also we can access individual customer record from customers/me endpoint,



We get the data of our john smith.

And all our endpoints are behaving consistently.

**Applying Model Permissions**:

By our current implementation only admin users can manage customers via the customers endpoint.

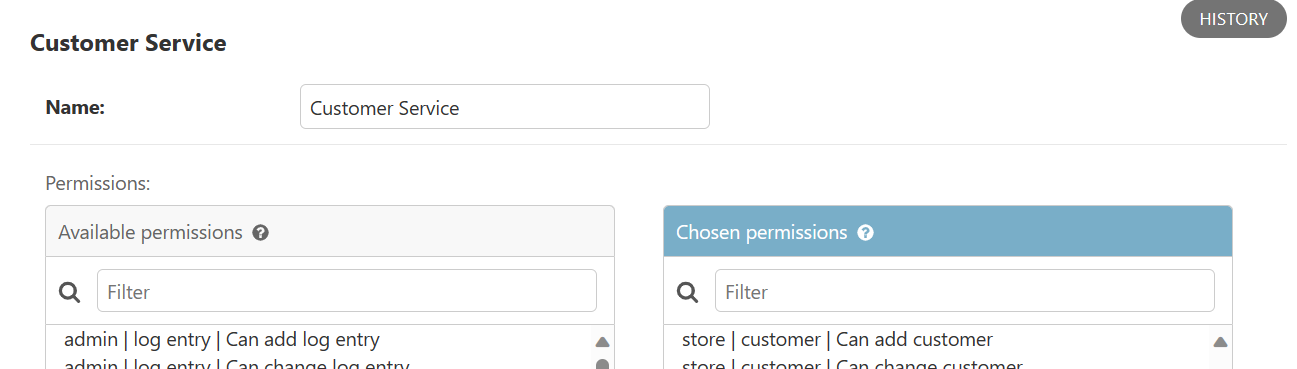
class CustomerViewSet(ModelViewSet):

    queryset = Customer.objects.all()

    serializer\_class = CustomerSerializer

    permission\_classes = [IsAdminUser] 🡪 *This one*

But earlier in the course we created a group called customer service.



What if you want to allow these people to manage our customers by our API as well?

This is where we use *Django Model Permissions*.

In our CustomerViewSet we are going to use another permission class called *DjangoModelPermissions*.

class CustomerViewSet(ModelViewSet):

    queryset = Customer.objects.all()

    serializer\_class = CustomerSerializer

    permission\_classes = [DjangoModelPermissions] 🡪 *Here*

    @action(detail=False, methods=["GET", "PUT"], permission\_classes=[IsAuthenticated])

    def me(self, request):

        (customer, created) = Customer.objects.get\_or\_create(user\_id=request.user.id)

        if request.method == "GET":

            serializer = CustomerSerializer(customer)

            return Response(serializer.data)

        elif request.method == "PUT":

            serializer = CustomerSerializer(customer, data=request.data)

            serializer.is\_valid(raise\_exception=True)

            serializer.save()

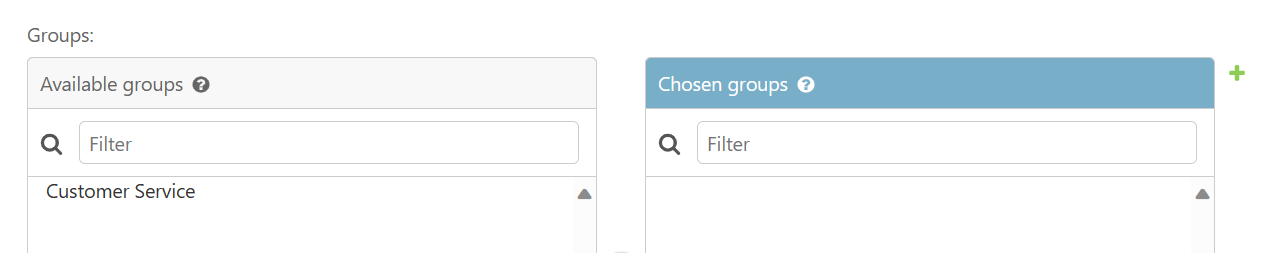
            return Response(serializer.data)

*When we apply this permission, the user has to be authenticated and they should have the relevant model permissions*.

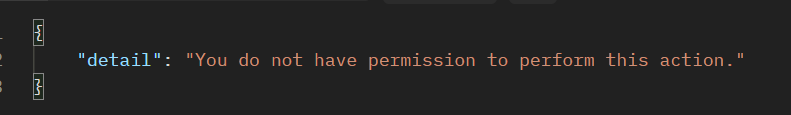
Our user *john.smith* has been assigned to *Customer Service* group, so we should be able to access customers end point when we are logged in as *john.smith*.



We can also create a new customer and update or delete existing ones. But if we remove john.smith from customer service group then this is what happens.



Now let us try updating (*PUT request*)phone number of a customer when logged in as John.smith



After removing permission I can only view data but cannot modify it. To understand why this is happening we will look the implementation of *DjangoModelPermissions* class.

class DjangoModelPermissions(BasePermission):

    """

    The request is authenticated using `django.contrib.auth` permissions.

    See: https://docs.djangoproject.com/en/dev/topics/auth/#permissions

    It ensures that the user is authenticated, and has the appropriate

    `add`/`change`/`delete` permissions on the model.

    This permission can only be applied against view classes that

    provide a `.queryset` attribute.

    """

    # Map methods into required permission codes.

    # Override this if you need to also provide 'view' permissions,

    # or if you want to provide custom permission codes.

    perms\_map = {

        'GET': [],

        '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'],

    }

    authenticated\_users\_only = True

    def get\_required\_permissions(self, method, model\_cls):

        """

        Given a model and an HTTP method, return the list of permission

        codes that the user is required to have.

        """

        kwargs = {

            'app\_label': model\_cls.\_meta.app\_label,

            'model\_name': model\_cls.\_meta.model\_name

        }

        if method not in self.perms\_map:

            raise exceptions.MethodNotAllowed(method)

        return [perm % kwargs for perm in self.perms\_map[method]]

    def \_queryset(self, view):

        assert hasattr(view, 'get\_queryset') \

            or getattr(view, 'queryset', None) is not None, (

            'Cannot apply {} on a view that does not set '

            '`.queryset` or have a `.get\_queryset()` method.'

        ).format(self.\_\_class\_\_.\_\_name\_\_)

        if hasattr(view, 'get\_queryset'):

            queryset = view.get\_queryset()

            assert queryset is not None, (

                '{}.get\_queryset() returned None'.format(view.\_\_class\_\_.\_\_name\_\_)

            )

            return queryset

        return view.queryset

    def has\_permission(self, request, view):

        # Workaround to ensure DjangoModelPermissions are not applied

        # to the root view when using DefaultRouter.

        if getattr(view, '\_ignore\_model\_permissions', False):

            return True

        if not request.user or (

           not request.user.is\_authenticated and self.authenticated\_users\_only):

            return False

        queryset = self.\_queryset(view)

        perms = self.get\_required\_permissions(request.method, queryset.model)

        return request.user.has\_perms(perms)

*perms\_map* is the dictionary of permissions. As you can see for GET, OPTIONS and HEAD we do not have any permissions.

    perms\_map = {

        'GET': [],

        'OPTIONS': [],

        'HEAD': [],

        'POST': ['%(app\_label)s.add\_%(model\_name)s'],

        'PUT': ['%(app\_label)s.change\_%(model\_name)s'],

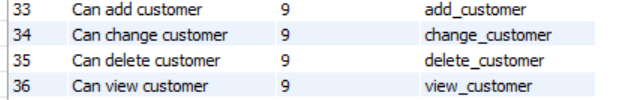
        'PATCH': ['%(app\_label)s.change\_%(model\_name)s'],

        'DELETE': ['%(app\_label)s.delete\_%(model\_name)s'],

    }

But to send a POST request, the user must have add\_ permission. First we have *app\_label* which will change dynamically at run time then *model\_name*. This is the code name for permission.

In the database look at the permission table.



Here we have codenames like *add\_customer*, *change\_customer* and so on. So what *we see in the perms\_map dictionary is the mapping between HTTP methods and relevant permissions*.

Similarly for sending PUT and PATCH request user needs to have the change permission.

To solve this problem and prevent people from outside customer service group from viewing data, we need to create a custom permission class and extend *DjangoModelPermissions*.

In the store app, go to permissions module and create a new permission class. We can call this *FullDjangoModelPermissions*.

class FullDjangoModelPermissions(permissions.DjangoModelPermissions):

    def \_\_init\_\_(self):

        self.perms\_map["GET"] = ["%(app\_label)s.view\_%(model\_name)s"]

Here we defined a constructor and set self.perms\_map[“GET”] (*override ‘GET’ method*) to have a *view* permission.

Now we will use this permission in Customer ViewSet.

class CustomerViewSet(ModelViewSet):

    queryset = Customer.objects.all()

    serializer\_class = CustomerSerializer

    permission\_classes = [FullDjangoModelPermissions] 🡪 *Change here*

    @action(detail=False, methods=["GET", "PUT"], permission\_classes=[IsAuthenticated])

    def me(self, request):

        (customer, created) = Customer.objects.get\_or\_create(user\_id=request.user.id)

        if request.method == "GET":

            serializer = CustomerSerializer(customer)

            return Response(serializer.data)

        elif request.method == "PUT":

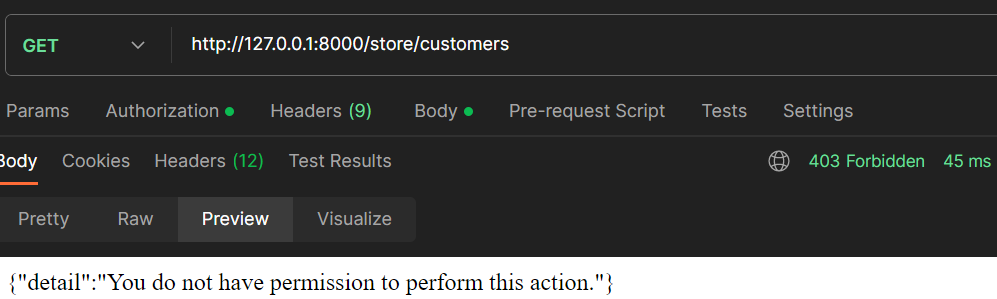
            serializer = CustomerSerializer(customer, data=request.data)

            serializer.is\_valid(raise\_exception=True)

            serializer.save()

            return Response(serializer.data)

Now if I try to access customers list from john.smith as a user.



We get 403 forbidden.

So even though we send our authentication credentials (*access token*), we do not have access to this resource.

We have another similar class called *DjangoModelPermissionsOrnonReadOnly*. This is exactly like DjangoModelPermissions but anonymous users will have read only access to private data.

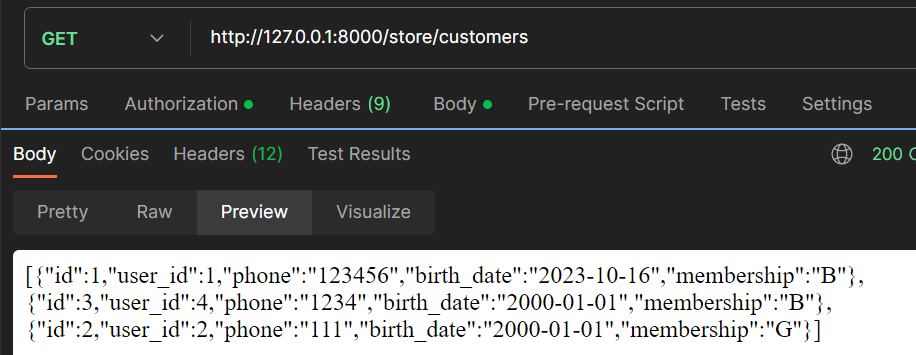
class CustomerViewSet(ModelViewSet):

    queryset = Customer.objects.all()

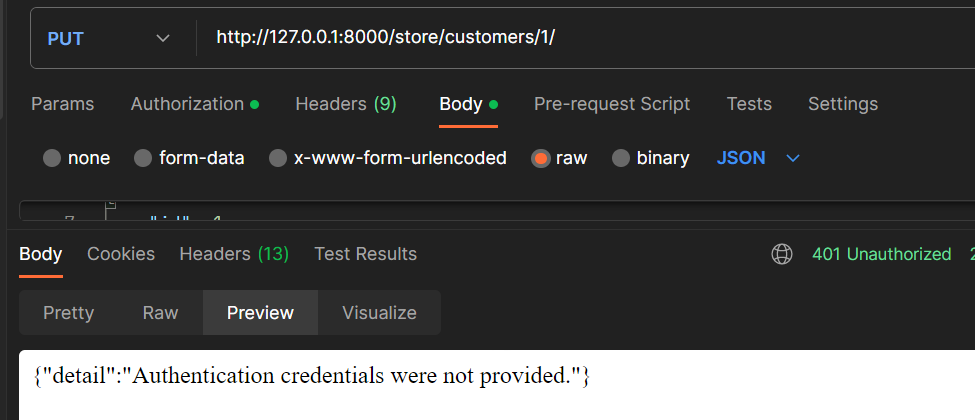
    serializer\_class = CustomerSerializer

    permission\_classes = [DjangoModelPermissionsOrAnonReadOnly]

Without access token, we can view all customers



But if I try to add a new customer or update one,



401 unauthorized.

So this is how model permissions work. But for this application it is a bit of overengineering. We do not really need that level of complexity.

For this application, we want to limit customer management operations only to admin users. So IsAdminUser permission class is enough for us.

class CustomerViewSet(ModelViewSet):

    queryset = Customer.objects.all()

    serializer\_class = CustomerSerializer

    permission\_classes = [IsAdminUser] 🡪 *Perfect for now*

**Applying custom model Permissions**:

The last thing we are going to talk about in this section is applying custom model permissions. In the previous section, we created a custom model permission called *cancel order*.

Now let us see how we can apply these custom model permissions to our API end points. For this lesson since currently we do not have orders API we are going to create a custom permission for our customer model.

In the Customer model, under Meta class, we should set permissions to a list of tuples.

Each tuple should include two values. First one is codename for permission(*like* ***view\_history****, which is the value we reference in our code*) and second value is a description for the permission (*like ‘Can view history’*).

class Customer(models.Model):

    MEMBERSHIP\_BRONZE = "B"

    MEMBERSHIP\_SILVER = "S"

    MEMBERSHIP\_GOLD = "G"

    MEMBERSHIP\_CHOICES = [

        (MEMBERSHIP\_BRONZE, "Bronze"),

        (MEMBERSHIP\_SILVER, "Silver"),

        (MEMBERSHIP\_GOLD, "Gold"),

    ]

    phone = models.CharField(max\_length=255)

    birth\_date = models.DateField(null=True)

    membership = models.CharField(

        max\_length=1, choices=MEMBERSHIP\_CHOICES, default=MEMBERSHIP\_BRONZE

    )

    user = models.OneToOneField(settings.AUTH\_USER\_MODEL, on\_delete=models.CASCADE)

    @admin.display(ordering="user\_\_first\_name")

    def first\_name(self):

        return self.user.first\_name

    @admin.display(ordering="user\_\_last\_name")

    def last\_name(self):

        return self.user.last\_name

    def \_\_str\_\_(self) -> str:

        return f"{self.user.first\_name} {self.user.last\_name}"

    class Meta:

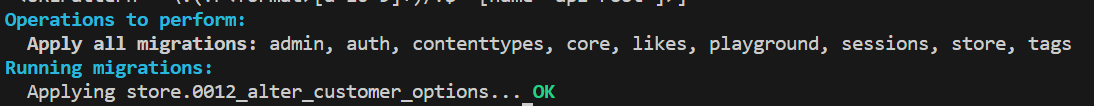
        ordering = ["user\_\_first\_name", "user\_\_last\_name"]

        permissions = [("view\_history", "Can view history")] *🡪 Here*

Now that we have modified the model, we need to create a migration and run it and as a result this permission will be stored in the database.

python manage.py makemigrations

python manage.py migrate



Next we need to go to customer viewset and create a custom action (*just like* ***me*** *action we created*) for viewing the history of a particular customer.

class CustomerViewSet(ModelViewSet):

    queryset = Customer.objects.all()

    serializer\_class = CustomerSerializer

    permission\_classes = [IsAdminUser]

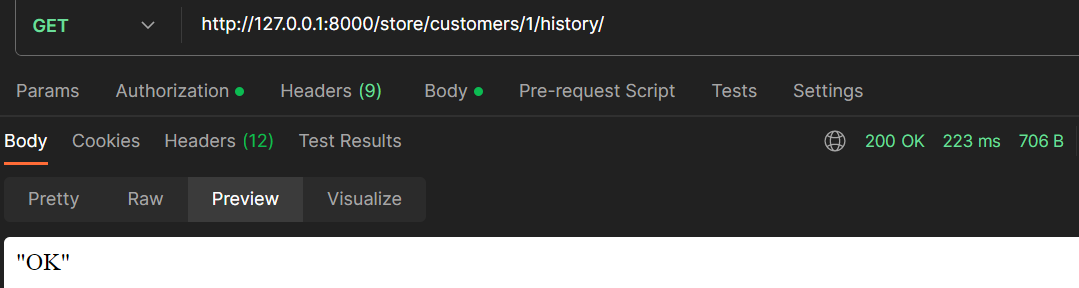
    @action(detail=True)

    def history(self, request, pk):

        return Response("OK")

So we defined a action called *history* with three parameters; self, request and pk (*pk, because this is for a particular customer*). In the action decorator we set the detail to True. For now we simply return a “OK” Response.

Now if we go to history endpoint,



We get a response.

Now we will apply the permissions, for that we need to create a new permission class in the *permissions* module of store app.

class ViewCustomerHistoryPermission(permissions.BasePermission):

    def has\_permission(self, request, view):

All permission classes that we create should inherit from BasePermission class. Here we override *has\_permission* like before but our condition is a little bit different.

class ViewCustomerHistoryPermission(permissions.BasePermission):

    def has\_permission(self, request, view):

        return request.user.has\_perm("store.view\_history")

Here we returned request.user.has\_perm(). This request.user object has a *has\_perm* method where we pass codename for the permission. It starts with app name (*store*), dot (**.**) then permission name (*view\_history*).

*If this returns true then this user is going to have permission and they will be able to access history*.

Final step is to go to our CustomerViewSet and decorate our history action with the new permission class we just created.

class CustomerViewSet(ModelViewSet):

    queryset = Customer.objects.all()

    serializer\_class = CustomerSerializer

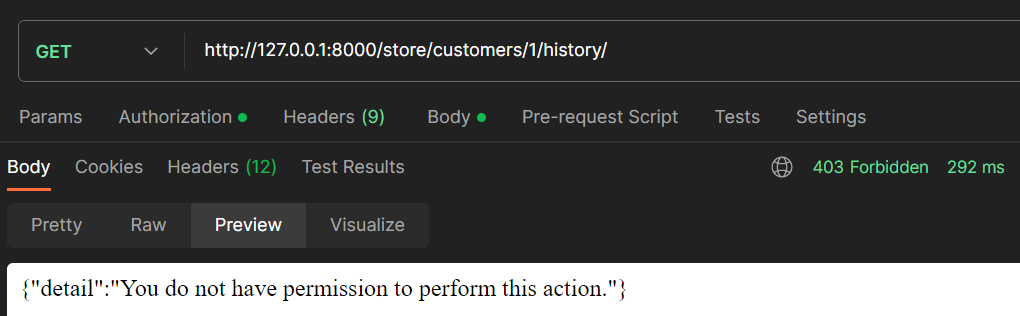
    permission\_classes = [IsAdminUser]

    @action(detail=True, permission\_classes=[ViewCustomerHistoryPermission])

    def history(self, request, pk):

        return Response("OK")

Since john.smith currently does not have this permission, he cannot access this endpoint.

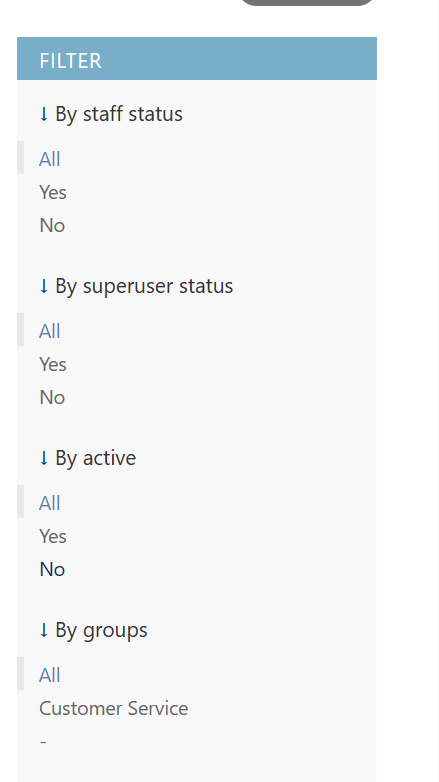


So let us go to admin panel and give john smith this explicit permission.



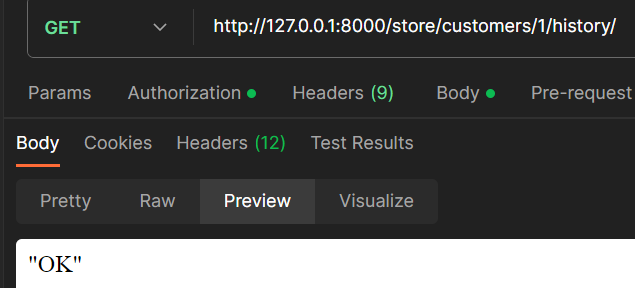
Note: We are adding this permission here. But remember this is just for this demo. As a best practice, we should never apply ad-hoc permissions because over time it becomes really hard to see who has what permission.

*Always create groups, add permissions to your groups and then add users to groups. This way we can easily filter users by groups to see who has what kind of permissions*.

🡨 on the right side we have a filter By groups. If you click on Customer Service group we can see all users assigned to this group. But we do not have filtering by permissions (*that is why avoid ad-hoc permissions*).

Back to our history endpoint,

Since john.smith has view history permission now.



We can see our message response now.