**Specifying a custom user model**

When you start your project with a custom user model, stop to consider if this is the right choice for your project

1. Keeping all user related information in one model , Add more complex database queries to retrieve related models.
2. Store Users information in model App Which related to .

That allows each app to specify its own user data requirements without conflicting by other apps

فى كل أب يتم تخزين المستخدمين له بكل الصلاحيات بتعتهم للإستخدام ذلك الأب مثال :

مجموعة (1) من المستخدمين هيستخدموا أب رقم (1) ولهم صلحيات( أ , ب) إذن داخل الموديل بتاع الأب نفسه هنعرف تلك المستخدمين بصلحياتهم كلها فقط

مجموعة (2) من المستخدمين هيستخدموا أب رقم (2) ولهم صلحيات( أ , ب ,ج,د) إذن داخل الموديل بتاع الأب نفسه هنعرف تلك المستخدمين بصلحياتهم كلها فقط

If you use the default authentication backend, then your model must have a single unique field that can be used for identification purposes. This can be a username, an email address

The easiest way to construct a compliant custom user model is to inherit from [**AbstractBaseUser**](https://docs.djangoproject.com/en/3.0/topics/auth/customizing/#django.contrib.auth.models.AbstractBaseUser)

[**AbstractBaseUser**](https://docs.djangoproject.com/en/3.0/topics/auth/customizing/#django.contrib.auth.models.AbstractBaseUser) provides the core implementation of a user model, including hashed passwords and tokenized password resets.

You must then provide some key implementation details:

**USERNAME\_FIELD**[**¶**](https://docs.djangoproject.com/en/3.0/topics/auth/customizing/#django.contrib.auth.models.CustomUser.USERNAME_FIELD)

A string describing the name of the field on the user model that is used as the unique identifier. This will usually be a username of some kind, but it can also be an email address, or any other unique identifier. The field *must* be unique (i.e., have **unique=True** set in its definition)

In the following example, the field **identifier** is used as the identifying field:

**class** **MyUser**(AbstractBaseUser):

identifier = models.CharField(max\_length=40, unique=**True**)

...

USERNAME\_FIELD = 'identifier'

**EMAIL\_FIELD**[**¶**](https://docs.djangoproject.com/en/3.0/topics/auth/customizing/#django.contrib.auth.models.CustomUser.EMAIL_FIELD)

A string describing the name of the email field on the **User** model. This value is returned by [**get\_email\_field\_name()**](https://docs.djangoproject.com/en/3.0/topics/auth/customizing/#django.contrib.auth.models.AbstractBaseUser.get_email_field_name).

**REQUIRED\_FIELDS**[**¶**](https://docs.djangoproject.com/en/3.0/topics/auth/customizing/#django.contrib.auth.models.CustomUser.REQUIRED_FIELDS)

A list of the field names that will be prompted for when creating a user via the [**createsuperuser**](https://docs.djangoproject.com/en/3.0/ref/django-admin/#django-admin-createsuperuser) management command. The user will be prompted to supply a value for each of these fields. It must include any field for which [**blank**](https://docs.djangoproject.com/en/3.0/ref/models/fields/#django.db.models.Field.blank) is **False** or undefined and may include additional fields you want prompted for when a user is created interactively. **REQUIRED\_FIELDS** has no effect in other parts of Django, like creating a user in the admin.

For example, here is the partial definition for a user model that defines two required fields - a date of birth and height:

**class** **MyUser**(AbstractBaseUser):

...

date\_of\_birth = models.DateField()

height = models.FloatField()

...

REQUIRED\_FIELDS = ['date\_of\_birth', 'height']

**is\_active**[**¶**](https://docs.djangoproject.com/en/3.0/topics/auth/customizing/#django.contrib.auth.models.CustomUser.is_active)

A boolean attribute that indicates whether the user is considered “active”. This attribute is provided as an attribute on **AbstractBaseUser** defaulting to **True**

**get\_full\_name()**[**¶**](https://docs.djangoproject.com/en/3.0/topics/auth/customizing/#django.contrib.auth.models.CustomUser.get_full_name)

Optional. A longer formal identifier for the user such as their full name. If implemented, this appears alongside the username in an object’s history in [**django.contrib.admin**](https://docs.djangoproject.com/en/3.0/ref/contrib/admin/#module-django.contrib.admin).

**get\_short\_name()**[**¶**](https://docs.djangoproject.com/en/3.0/topics/auth/customizing/#django.contrib.auth.models.CustomUser.get_short_name)

Optional. A short, informal identifier for the user such as their first name. If implemented, this replaces the username in the greeting to the user in the header of [**django.contrib.admin**](https://docs.djangoproject.com/en/3.0/ref/contrib/admin/#module-django.contrib.admin).

The following attributes and methods are available on any subclass of [**AbstractBaseUser**](https://docs.djangoproject.com/en/3.0/topics/auth/customizing/#django.contrib.auth.models.AbstractBaseUser):

**get\_username()**[**¶**](https://docs.djangoproject.com/en/3.0/topics/auth/customizing/#django.contrib.auth.models.AbstractBaseUser.get_username)

Returns the value of the field nominated by **USERNAME\_FIELD**.

**clean()**[**¶**](https://docs.djangoproject.com/en/3.0/topics/auth/customizing/#django.contrib.auth.models.AbstractBaseUser.clean)

Normalizes the username by calling [**normalize\_username()**](https://docs.djangoproject.com/en/3.0/topics/auth/customizing/#django.contrib.auth.models.AbstractBaseUser.normalize_username). If you override this method, be sure to call **super()** to retain the normalization.

**get\_email\_field\_name()**[**¶**](https://docs.djangoproject.com/en/3.0/topics/auth/customizing/#django.contrib.auth.models.AbstractBaseUser.get_email_field_name)

Returns the name of the email field specified by the [**EMAIL\_FIELD**](https://docs.djangoproject.com/en/3.0/topics/auth/customizing/#django.contrib.auth.models.CustomUser.EMAIL_FIELD) attribute. Defaults to **'email'** if **EMAIL\_FIELD** isn’t specified.

**normalize\_username(*username*)**[**¶**](https://docs.djangoproject.com/en/3.0/topics/auth/customizing/#django.contrib.auth.models.AbstractBaseUser.normalize_username)

Applies NFKC Unicode normalization to usernames so that visually identical characters with different Unicode code points are considered identical.

**is\_authenticated**[**¶**](https://docs.djangoproject.com/en/3.0/topics/auth/customizing/#django.contrib.auth.models.AbstractBaseUser.is_authenticated)

Read-only attribute which is always **True** (as opposed to **AnonymousUser.is\_authenticated** which is always **False**). This is a way to tell if the user has been authenticated

**is\_anonymous**[**¶**](https://docs.djangoproject.com/en/3.0/topics/auth/customizing/#django.contrib.auth.models.AbstractBaseUser.is_anonymous)

Read-only attribute which is always **False**. This is a way of differentiating [**User**](https://docs.djangoproject.com/en/3.0/ref/contrib/auth/#django.contrib.auth.models.User) and [**AnonymousUser**](https://docs.djangoproject.com/en/3.0/ref/contrib/auth/#django.contrib.auth.models.AnonymousUser) objects. Generally, you should prefer using [**is\_authenticated**](https://docs.djangoproject.com/en/3.0/ref/contrib/auth/#django.contrib.auth.models.User.is_authenticated) to this attribute.

**Writing a manager for a custom user model**[**¶**](https://docs.djangoproject.com/en/3.0/topics/auth/customizing/#writing-a-manager-for-a-custom-user-model)

You should also define a custom manager for your user model. If your user model defines **username**, **email**, **is\_staff**, **is\_active**, **is\_superuser**, **last\_login**, and **date\_joined** fields the same as Django’s default user, you can install Django’s [**UserManager**](https://docs.djangoproject.com/en/3.0/ref/contrib/auth/#django.contrib.auth.models.UserManager); however, if your user model defines different fields, you’ll need to define a custom manager that extends [**BaseUserManager**](https://docs.djangoproject.com/en/3.0/topics/auth/customizing/#django.contrib.auth.models.BaseUserManager) providing two additional methods

**create\_user(*username\_field*, *password=None*, *\*\*other\_fields*)**[**¶**](https://docs.djangoproject.com/en/3.0/topics/auth/customizing/#django.contrib.auth.models.CustomUserManager.create_user)

The prototype of **create\_user()** should accept the username field, plus all required fields as arguments. For example, if your user model uses **email** as the username field, and has **date\_of\_birth** as a required field, then **create\_user** should be defined as:

**def** create\_user(self, email, date\_of\_birth, password=**None**):

*# create user here*

...

**create\_superuser(*username\_field*, *password=None*, *\*\*other\_fields*)**[**¶**](https://docs.djangoproject.com/en/3.0/topics/auth/customizing/#django.contrib.auth.models.CustomUserManager.create_superuser)

The prototype of **create\_superuser()** should accept the username field, plus all required fields as arguments. For example, if your user model uses **email** as the username field, and has **date\_of\_birth** as a required field, then **create\_superuser** should be defined as:

**def** create\_superuser(self, email, date\_of\_birth, password=**None**):

*# create superuser here*

...

[**BaseUserManager**](https://docs.djangoproject.com/en/3.0/topics/auth/customizing/#django.contrib.auth.models.BaseUserManager) provides the following utility methods:

**normalize\_email(*email*)**[**¶**](https://docs.djangoproject.com/en/3.0/topics/auth/customizing/#django.contrib.auth.models.BaseUserManager.normalize_email)

Normalizes email addresses by lowercasing the domain portion of the email address.

.After Do This In Settings File Must Add The Following Code That Specific User Model :

AUTH\_USER\_MODEL="profiles\_rest\_api.UserProfile"

Then Run makemigrations ,migrate command (Must After Create Model Custom Users) Not Before Error

ViewSet Benfites:

1. Perfect for standard database Operations
2. Fastest way to make a database interface

When To Use ViewSets??

1. A simple CRUD interface to you database
2. A quick and simple API
3. Littel to no customization on the logic
4. Working with standard data structures

2 Way To Use Viewsets

1. Override All Action (by inherit from viewsets.ViewSet )

class UserViewSet(viewsets.ViewSet):

"""

Example empty viewset demonstrating the standard

actions that will be handled by a router class.

If you're using format suffixes, make sure to also include

the `format=None` keyword argument for each action.

"""

def list(self, request):

pass

def create(self, request):

pass

def retrieve(self, request, pk=None):

pass

def update(self, request, pk=None):

pass

def partial\_update(self, request, pk=None):

pass

def destroy(self, request, pk=None):

pass

1. Using Default Action behaviour That Providers from classes (by inherit from viewsets.ModelViewSet ) :

class UserViewSet(viewsets.ModelViewSet):

"""

A viewset for viewing and editing user instances.

"""

serializer\_class = UserSerializer

queryset = User.objects.all()

Create Serializers that Handle CustomUser

class UserProfileSerailzer(serializers.ModelSerializer):

    """Serializer a user profile object """

    class Meta:

        model=UserProfile

        #fields='\_\_all\_\_'

        #fields=('id','email','name','password','is\_superuser')

        fields=('id','email','name','password')

        extra\_kwargs={

            'password':{

                'write\_only':True ,

                'style':{'input\_type':'password'}

            }

        }

    def create(self,validated\_data):

        """Create And rteurn a new user """

        user = UserProfile.objects.create\_user(

            email=validated\_data['email'],

            name=validated\_data['name'],

            password=validated\_data['password']

        )

        return user

Hint:

- fields=’\_\_all\_\_’

All fields will appear

- fields=('id','email','name','password','is\_superuser')

This only fields will apppera

-extra-kwargs={}

الخصائص اللى هتضيفها على الحقول اللى هتظهر مثلا الباسور ميظهرش فى العرض و لما تسجل يتكتب نوع باسورد يظهر نقط فقط

Must override create method

# [Authentication](https://www.django-rest-framework.org/api-guide/authentication/#authentication)

Authentication is the mechanism of associating an incoming request with a set of identifying credentials, such as the user the request came from, or the token that it was signed with

REST framework provides a number of authentication schemes out of the box, and also allows you to implement custom schemes.

Authentication is always run at the very start of the view, before the permission and throttling checks occur, and before any other code is allowed to proceed.

The request.user property will typically be set to an instance of the contrib.auth package's User class.

The request.auth property is used for any additional authentication information, for example, it may be used to represent an authentication token that the request was signed with

## [TokenAuthentication](https://www.django-rest-framework.org/api-guide/authentication/#tokenauthentication)

This authentication scheme uses a simple token-based HTTP Authentication scheme. Token authentication is appropriate for client-server setups, such as native desktop and mobile clients.

To use the TokenAuthentication scheme you'll need to [configure the authentication classes](https://www.django-rest-framework.org/api-guide/authentication/#setting-the-authentication-scheme) to include TokenAuthentication, and additionally include rest\_framework.authtoken in your INSTALLED\_APPS setting:

INSTALLED\_APPS = [

...

'rest\_framework.authtoken'

]

**Note:** Make sure to run manage.py migrate after changing your settings. The rest\_framework.authtoken app provides Django database migrations.

Create Loging functionlty to our project:

In this using

tokenauthentication type :

ده عبارة عن راندم استرنج بيتحط فى الهيدر مع الركوست بتاع اليوزر عند طلب اى شئ

# [Permissions](https://www.django-rest-framework.org/api-guide/permissions/#permissions)

Permissions are used to grant or deny access for different classes of users to different parts of the API.

Permission checks are always run at the very start of the view, before any other code is allowed to proceed. Permission checks will typically use the authentication information in the request.user and request.auth properties to determine if the incoming request should be permitted.

# [How permissions are determined](https://www.django-rest-framework.org/api-guide/permissions/#how-permissions-are-determined)

Permissions in REST framework are always defined as a list of permission classes.

Before running the main body of the view each permission in the list is checked. If any permission check fails an exceptions.PermissionDenied or exceptions.NotAuthenticated exception will be raised, and the main body of the view will not run.

# [Custom permissions](https://www.django-rest-framework.org/api-guide/permissions/#custom-permissions)

To implement a custom permission, override BasePermission and implement either, or both, of the following methods:

* .has\_permission(self, request, view)
* .has\_object\_permission(self, request, view, obj)

The methods should return True if the request should be granted access, and False otherwise.

If you need to test if a request is a read operation or a write operation, you should check the request method against the constant SAFE\_METHODS, which is a tuple containing 'GET', 'OPTIONS' and 'HEAD'. For example:

if request.method in permissions.SAFE\_METHODS:

# Check permissions for read-only request

else:

# Check permissions for write request

**Note**: The instance-level has\_object\_permission method will only be called if the view-level has\_permission checks have already passed. Also note that in order for the instance-level checks to run, the view code should explicitly call .check\_object\_permissions(request, obj). If you are using the generic views then this will be handled for you by default. (Function-based views will need to check object permissions explicitly, raising PermissionDenied on failure.)

Custom permissions will raise a PermissionDenied exception if the test fails. To change the error message associated with the exception, implement a message attribute directly on your custom permission. Otherwise the default\_detail attribute from PermissionDenied will be used.

from rest\_framework import permissions

class CustomerAccessPermission(permissions.BasePermission):

message = 'Adding customers not allowed.'

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

...

## [Examples](https://www.django-rest-framework.org/api-guide/permissions/#examples)

The following is an example of a permission class that checks the incoming request's IP address against a blacklist, and denies the request if the IP has been blacklisted.

from rest\_framework import permissions

class BlacklistPermission(permissions.BasePermission):

"""

Global permission check for blacklisted IPs.

"""

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

ip\_addr = request.META['REMOTE\_ADDR']

blacklisted = Blacklist.objects.filter(ip\_addr=ip\_addr).exists()

return not blacklisted

As well as global permissions, that are run against all incoming requests, you can also create object-level permissions, that are only run against operations that affect a particular object instance. For example:

class IsOwnerOrReadOnly(permissions.BasePermission):

"""

Object-level permission to only allow owners of an object to edit it.

Assumes the model instance has an `owner` attribute.

"""

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

# Read permissions are allowed to any request,

# so we'll always allow GET, HEAD or OPTIONS requests.

if request.method in permissions.SAFE\_METHODS:

return True

# Instance must have an attribute named `owner`.

return obj.owner == request.user

Note that the generic views will check the appropriate object level permissions, but if you're writing your own custom views, you'll need to make sure you check the object level permission checks yourself. You can do so by calling self.check\_object\_permissions(request, obj) from the view once you have the object instance. This call will raise an appropriate APIException if any object-level permission checks fail, and will otherwise simply return.

Also note that the generic views will only check the object-level permissions for views that retrieve a single model instance. If you require object-level filtering of list views, you'll need to filter the queryset separately. See the [filtering documentation](https://www.django-rest-framework.org/api-guide/filtering/) for more details.

In my project :

has\_object\_permission(self,request,view,obj)

دى بتستخدمها لما يكون اليوزر هيعدل على اوبجكت فى الموقع

Self: for python create method in class

Request: store information about user

View : this is view that response for user request

Obj: object this user want to change ,(do something on it)

If this method return True , User Request will be perform

If this method return False , User Request Not Apply

لما اليوزر يطلب ريكوست اول حاجة بتتنفذ داخل الفيو هو ال (authentication\_classes)بالترتيب اللى محطوط داخل التوبل

بعد ده يتنفذ ال(permission\_classes) بالترتيب ايضا

From This Course :

1. Create , Update , delete , & manage user profiles
2. Authenticate with username & password
3. Manage user profile feed item