2 . To compress data in a Django CRUD view, you can use Django's built-in compression middleware. This middleware can automatically compress response content for browsers

3. Query parameters included in the URL and allows you to pass additional information to a view in Django. They are commonly used for filtering and searching a view.

4. HttpResponse is a class provided by Django for generating HTTP responses. It allows you to return a response with content, status code, and headers. The renderers used by the Response class can't be used in django to use response I need to serialize the data into primitive datatypes before creating the Response object.

5. Class-based views (CBVs) and function-based views (FBVs) are two ways to define views in Django. Function-based views are Python functions. They take a request as an argument and return an HTTP response. They are easy to use for one specific HTTP method. Class-based views are classes. They have methods to handle various HTTP methods. They are good to use for more complex views, especially when the same view needs to respond to different HTTP methods.

6. To make an FBV handle multiple HTTP methods, I can use conditional statements to each request . for example, I can use if statement to see which request is it .

7. when I create an object without his primary key the `save()` function determines to create a new object. the `save()` function update an existing object when I retrieve an existing object from the database and the primary key is already set.

8. `full\_clean()` method is good To handle errors when the validation didn't work when saving a model instance .This function allows to catch validation errors before attempting to save the object, ensuring that you don't save an invalid object to the database.

9. this line performs validation on the serializer instance and raises an exception if the data being validated is not valid. this way is better than the original way because it allows to catch validation errors without having to write custom logic for error handling.

11. To update only one field in a model using a POST request, the request need to be sent with the field you want to update and its new value, and then you update that specific field while keeping the other fields unchanged. An example: class UpdateBookTitleView(APIView):

    def post(self, request, book\_id):

        try:

            book = Book.objects.get(id=book\_id)

        except Book.DoesNotExist:

            return Response({'error': 'Book not found'}, status=status.HTTP\_404\_NOT\_FOUND)

        new\_title = request.data.get('title')

12. In DRF, depth option is used to control the depth of serialization when dealing with related models. By default, serializers only include the primary keys of related objects. the `depth` value can be increase to include deeper levels of related objects in the serialization.

14. read\_only\_fields attribute is used in a serializer class to specify which fields should be for read-only.

15. The `SerializerMethodField` allows to include custom fields or manipulating existing fields in the serialization process

16. SlugRelatedField is a field that allows you to represent relationships using a string representation of the related object. This is particularly useful when the related object has many fields, and you only need a specific field or a few fields from it. Using read\_only=True means that this field is read-only and can only be used for displaying data.

from rest\_framework import serializers

class MySerializer(serializers.ModelSerializer):

category = serializers.SlugRelatedField(

queryset=Category.objects.all(),

slug\_field='name',

read\_only=True

)

class Meta:

model = Product

fields = ('id', 'name', 'category')

17. select related method is used for ForeignKey and OneToOneField relationships. It performs a SQL join to retrieve related object data along with the main object data in a single query.

    books = Product.objects.select\_related('genre')

    for book in books:

        print(book.name, product.genre.name)

prefetch related method is used for ManyToManyField and reverse ForeignKey relationships. It retrieves the related objects separately.

genres= ganre.objects.prefetch\_related('books')

for ganre in ganres:

    print(ganre.name)

    for book in ganre.books.all():

        print(book.name)

18. To retrieve all the information about a user, including all the messages they have written, we can make a nested serializer within the UserSerializer for the related messages.

class UserDetailView(APIView):

def get(self, request, user\_id):

user = User.objects.get(pk=user\_id)

serializer = UserSerializer(user)

return Response(serializer.data)

19. Writable nested serializers allows you to work with complex nested data structures, such as nested objects, lists of objects, or child objects that are associated with a parent object. It enables you to perform actions like creating a parent object and its associated child objects in a single API call. Parent Serialize is the main serializer responsible for handling the parent object. child serializer defines how to serialize and deserialize related child objects.

There are several additional packages, libraries, and projects that offer solutions and extensions for handling complex scenarios that DRF is not providing. for example, The Django-Filter package allows the creation of complex query filters for your DRF views, providing powerful filtering options for API endpoints and DRF extensions provide a set of useful extensions and utilities for the Django Rest Framework. It includes features like nested serializer fields, filtering, and more.