DRF- Django Rest API

Steps to create restapi in django

1. **Install djangorestframework -** pip install djangorestframework
2. Create one new app - django-admin startapp appname
3. Include rest_api and appname in installed_apps in settings.py
4. Create models to store data
5. Run migration
6. Create serializers

Create a serializer in `myapp/serializers.py` to convert model instances to JSON:

```
# myapp/serializers.py
from rest_framework import serializers
from .models import Task

class TaskSerializer(serializers.ModelSerializer):
    class Meta:
        model = Task
        fields = '__all__'
```

7. Create views in `myapp/views.py` using Django REST Framework viewsets:

```
# myapp/views.py
from rest_framework import viewsets
from .models import Task
from .serializers import TaskSerializer

class TaskViewSet(viewsets.ModelViewSet):
    queryset = Task.objects.all()
    serializer_class = TaskSerializer
```

8. Create a `myapp/urls.py` file to configure URLs:

```
from django.urls import path, include
router = DefaultRouter()
router.register(r'tasks', TaskViewSet, basename='task')
from rest_framework.routers import DefaultRouter
from .views import TaskViewSet
urlpatterns = [
    path('api/', include(router.urls)),
]
```

1. Serializers

Serialization is the process of converting complex data types like Django models into simpler representations, typically JSON.

```
# serializers.py
from rest_framework import serializers
from .models import Task

class TaskSerializer(serializers.ModelSerializer):
    class Meta:
        model = Task
        fields = '__all__'
```

2. Pagination:

Pagination is the process of dividing a large set of results into smaller chunks (pages). DRF provides built-in support for pagination.

We need to configure in our settings.py

```
REST_FRAMEWORK = {
    'DEFAULT_PAGINATION_CLASS': 'rest_framework.pagination.PageNumberPagination',
    'PAGE_SIZE': 10,
}
```

Per-View Configuration:

```
from rest_framework.pagination import PageNumberPagination

class YourModelViewSet(viewsets.ModelViewSet):
    queryset = YourModel.objects.all()
    serializer_class = YourModelSerializer
    pagination_class = PageNumberPagination
    page_size = 10
```

We can even have custom pagination

```
class CustomPageNumberPagination(PageNumberPagination):
    page_size = 5
    page_size_query_param = 'page_size'
    max_page_size = 100

class TaskListAPIView(ListAPIView):
    queryset = Task.objects.all()
    serializer_class = TaskSerializer
    pagination_class = CustomPageNumberPagination
```

The API will return paginated results, showing 5 tasks per page by default. Clients can customize the page size using the `page_size` query parameter. For example:

- http://localhost:8000/api/tasks/?page_size=10

4. Authentication and Permissions:

Authentication is the process of verifying the identity of a user, device, or system. It ensures that the entity trying to access a resource is who it claims to be.

I. Token-based Authentication:
II. Session-based Authentication:
III. OAuth and OAuth2:
    - Allows users to grant third-party applications limited access to their resources without exposing credentials.
IV. API Keys:
    - Users or applications include an API key in their requests.
    - The server verifies the API key to authenticate the user or application.

Permissions:

Permissions control what actions users or clients are allowed to perform on specific resources. Permissions are typically associated with roles assigned to users. Common permission levels include:

How to send request to server using curl

1. GET Request (Retrieve Tasks):

curl -H "Authorization: ApiKey YOUR_API_KEY" https://api.example.com/tasks/

Alternative: curl --header "Authorization: ApiKey YOUR_API_KEY" https://api.example.com/tasks/

2. POST Request (Create a New Task):

curl -H "Authorization: ApiKey YOUR_API_KEY" -H "Content-Type: application/json" -X POST -d '{"title": "New Task", "description": "Task Description", "due_date": "2024-12-31"}' https://api.example.com/tasks/

Alternative: curl --header "Authorization: ApiKey YOUR_API_KEY" --header "Content-Type: application/json" --request POST --data '{"title": "New Task", "description": "Task Description", "due_date": "2024-12-31"}' https://api.example.com/tasks/

3. PUT Request (Update an Existing Task):

curl -H "Authorization: ApiKey YOUR_API_KEY" -H "Content-Type: application/json" -X PUT -d '{"title": "Updated Task", "description": "Updated Description", "due_date": "2025-01-15"}' https://api.example.com/tasks/1/

Alternative: curl --header "Authorization: ApiKey YOUR_API_KEY" --header "Content-Type: application/json" --request PUT --data '{"title": "Updated Task", "description": "Updated Description", "due_date": "2025-01-15"}' https://api.example.com/tasks/1/

4. DELETE Request (Delete a Task):

curl -H "Authorization: ApiKey YOUR_API_KEY" -X DELETE https://api.example.com/tasks/1/

```
# Alternative: curl --header "Authorization: ApiKey YOUR_API_KEY" --request DELETE https://api.example.com/tasks/1/
```

5. Filtering, Searching, and Ordering:

1. Filtering:

Filtering allows clients to request a subset of resources based on specific criteria. In this example, we'll implement filtering based on the `due_date` field.

```
# views.py
from rest_framework import generics
from .models import Task
from .serializers import TaskSerializer

class TaskListAPIView(generics.ListAPIView):
    queryset = Task.objects.all()
    serializer_class = TaskSerializer

    def get_queryset(self):
        queryset = super().get_queryset()
        due_date = self.request.query_params.get('due_date', None)
        if due_date:
            queryset = queryset.filter(due_date=due_date)
        return queryset
```

Now, clients can filter tasks by providing a `due_date` parameter in the URL:

```
curl https://api.example.com/tasks/?due_date=2024-12-31
```

2. Searching:

Searching allows clients to perform a full-text search on specific fields. We'll enable searching on the `title` field.

```
# views.py
from rest_framework import generics
```

```
from .models import Task
from .serializers import TaskSerializer

class TaskListAPIView(generics.ListAPIView):
    queryset = Task.objects.all()
    serializer_class = TaskSerializer
    search_fields = ['title']  # Enable searching on the 'title' field

    def get_queryset(self):
        queryset = super().get_queryset()
        return queryset
```

Now, clients can perform a search by providing a `search` parameter in the URL:

curl https://api.example.com/tasks/?search=example

3. Ordering:

Ordering allows clients to specify the order in which the results should be returned. We'll enable ordering on the `due_date` field

```
from rest_framework import generics
from .models import Task
from .serializers import TaskSerializer
class TaskListAPIView(generics.ListAPIView):
        queryset = Task.objects.all()
        serializer_class = TaskSerializer
        ordering_fields = ['due_date']  # Enable ordering on the 'due_date' field
        def get_queryset(self):
                queryset = super().get_queryset()
                return queryset
```

Now, clients can order tasks by providing an `ordering` parameter in the URL:

curl https://api.example.com/tasks/?ordering=due_date

Types of Views in djangorestframework

1. APIView:

- Purpose:

- Base class for creating custom views that handle HTTP methods directly.
- Allows you to define methods like `get`, `post`, `put`, **etc., for handling specific HTTP methods.**

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

class MyCustomView(APIView):
    def get(self, request, *args, **kwargs):
        # Your logic here
        return Response({'message': 'GET request processed'}, status=status.HTTP_200_OK)
```

2. GenericAPIView:

- Purpose:
  - Extends `APIView` **and provides additional features for handling common patterns in API development.**
  - **Includes mixins like** `ListModelMixin`, `CreateModelMixin`, `UpdateModelMixin`,destroymixin **etc., to handle common CRUD operations.**

genericAPIView used mixins for common crud operations

```
from rest_framework.generics import GenericAPIView
from rest_framework.mixins import ListModelMixin
from .models import MyModel
from .serializers import MyModelSerializer
class TaskListCreateAPIView(GenericAPIView, ListModelMixin, CreateModelMixin):
    queryset = Task.objects.all()
    serializer_class = TaskSerializer

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

Here get method using ListModelMixin for listing tasks
Post method uses CreateModelMixin for creating new tasks

3. ModelViewSet:

- Purpose:
  - Extends `GenericAPIView` **and includes additional features specifically tailored for handling models.**

- **Provides built-in support for common CRUD operations (list, create, retrieve, update, destroy) with minimal configuration.**

```
from rest_framework.viewsets import ModelViewSet
from .models import MyModel
from .serializers import MyModelSerializer

class MyModelViewSet(ModelViewSet):
    queryset = MyModel.objects.all()
    serializer_class = MyModelSerializer
```

When dealing with very large datasets in Django's ORM (Object-Relational Mapping), it's important to optimize your queries and use efficient techniques to avoid performance issues. Here are some strategies to implement ORM queries for very large data:

1. **Use `values()` or `values_list()` to Fetch Specific Fields:**

When querying large datasets, fetching only the fields you need can significantly reduce the amount of data transferred. Use `values()` or `values_list()` to select only the necessary fields.

```
large_data = MyModel.objects.values('field1', 'field2')
# or
large_data = MyModel.objects.values_list('field1', 'field2')
```

2. **Use `iterator()` for QuerySets:**

When dealing with a large dataset, consider using the `iterator()` method. It fetches a small number of rows at a time, reducing memory consumption.

```python
large_data = MyModel.objects.all().iterator()
for item in large_data:
    # Process each item
    pass
```

3. **Batch Processing with `count()` and `limit()` in Chunks:**

For very large datasets, processing records in chunks can be more efficient. Use `count()` to get the total number of records and then fetch and process them in chunks using `limit()` and `offset()`.

```
chunk_size = 1000
total_records = MyModel.objects.count()
for offset in range(0, total_records, chunk_size):
    chunk = MyModel.objects.all().order_by('id')[offset:offset + chunk_size]
    for item in chunk:
        # Process each item
        pass
```

4. **Use `select_related()` and `prefetch_related()` Wisely:**

Optimize queries by using `select_related()` and `prefetch_related()` to fetch related objects efficiently, reducing the number of queries.

```
large_data = MyModel.objects.select_related('related_model')
# or
large_data = MyModel.objects.prefetch_related('related_model_set')
```

5. **Indexing:**

Ensure that your database tables have appropriate indexes on columns used in filtering and ordering, as this can significantly speed up queries.

6. **Use Database-Specific Optimizations:**

Leverage database-specific optimizations. Each database has its own features and optimizations. For example, PostgreSQL has powerful indexing and optimization features.

7. **Consider Raw SQL for Complex Queries:**

For complex queries that can't be easily expressed using the ORM, consider using raw SQL queries while still being cautious about SQL injection risks.

8. **Caching:**

Consider caching the results of queries if they are relatively static. Django provides a caching framework that can be used to store query results.

Remember to profile your queries, monitor database performance, and adjust your approach based on the characteristics of your specific dataset and the database engine you are using.

select_related() and prefetch_related()

are powerful tools in Django's ORM for optimizing database queries, especially when dealing with related objects and avoiding the N+1 query problem. Here's how you can use them:

`select_related()`:

`select_related()` performs a SQL join to include related object data in the same query. It's useful when you have a ForeignKey or OneToOneField relationship, and you want to fetch the related object data without additional queries.

```
class Author(models.Model):
    name = models.CharField(max_length=100)

class Book(models.Model):
    title = models.CharField(max_length=200)
    author = models.ForeignKey(Author, on_delete=models.CASCADE)
```

Usage in a query:

```
# Without select_related (N+1 queries problem):
books = Book.objects.all()
for book in books:
    print(book.author.name)  # Causes a separate query for each book's author

# With select_related:
books = Book.objects.select_related('author').all()
for book in books:
    print(book.author.name)  # No additional queries, related author data is fetched in the same query
```

`prefetch_related()`:

`prefetch_related()` is used for ManyToManyField or reverse ForeignKey/OneToOneField relationships. It fetches the related objects separately and does a lookup to match them efficiently in Python.

```
class Category(models.Model):
    name = models.CharField(max_length=100)

class Article(models.Model):
    title = models.CharField(max_length=200)
    categories = models.ManyToManyField(Category)
```

Usage in a query:

```
# Without prefetch_related (N+1 queries problem):
articles = Article.objects.all()
for article in articles:
    categories = article.categories.all()  # Causes a separate query for each article's categories

# With prefetch_related:
articles = Article.objects.prefetch_related('categories').all()
for article in articles:
    categories = article.categories.all()  # No additional queries, related categories data is fetched efficiently
```

The difference is that:

- **`select_related` does an SQL join and therefore gets the results back as part of the table from the SQL server**
- **`prefetch_related` on the other hand executes another query and therefore reduces the redundant columns in the original object (`ModelA` in the above example)**

HTTP status codes

are grouped into different categories, each representing a specific class of responses. Here's an overview of the main status code categories along with some common status codes within each category:

1. **1xx - Informational:**
   - 100: Continue
   - 101: Switching Protocols
   - 102: Processing

2. **2xx - Success:**
   - 200: OK
   - 201: Created
   - 202: Accepted
   - 204: No Content
   - 206: Partial Content

3. **3xx - Redirection:**
   - 300: Multiple Choices
   - 301: Moved Permanently
   - 302: Found (Moved Temporarily)
   - 304: Not Modified
   - 307: Temporary Redirect
   - 308: Permanent Redirect

4. **4xx - Client Error:**
   - 400: Bad Request
   - 401: Unauthorized
   - 403: Forbidden
   - 404: Not Found
   - 405: Method Not Allowed
   - 409: Conflict
   - 429: Too Many Requests

5. **5xx - Server Error:**
   - 500: Internal Server Error
   - 501: Not Implemented
   - 502: Bad Gateway
   - 503: Service Unavailable
   - 504: Gateway Timeout
   - 505: HTTP Version Not Supported

Each status code carries specific meanings, indicating the outcome or status of an HTTP request-response cycle. For instance:
- `1xx: Informational – Communicates transfer protocol-level information.
- `2xx` codes signify successful requests.
- `3xx` codes indicate redirection.
- `4xx` codes denote client-side errors (e.g., not found, unauthorized).
- `5xx` codes indicate server-side errors (e.g., server failure).

These status codes play a crucial role in communication between clients and servers, providing information about the result or status of a request and guiding the handling of responses by the client application or user.

HTTP (Hypertext Transfer Protocol) is a protocol used for transmitting data over the internet. It serves as the foundation for communication between web clients (such as web browsers) and servers, enabling the transfer of various types of data, including text, images, videos, and more.

Key aspects of HTTP:

HTTP (Hypertext Transfer Protocol) and **HTTPS (Hypertext Transfer Protocol Secure)** are both protocols used for transferring data over the internet, but they differ in terms of security and the way data is transmitted.

HTTP (Hypertext Transfer Protocol):
- **Security:** HTTP is not secure. Data transmitted over HTTP is sent in plaintext, making it susceptible to interception and potential manipulation by attackers.
- **Protocol:** It is the standard protocol used for transmitting data between a web server and a web browser.
- **Port:** HTTP uses port 80 by default for communication.
- **URL:** URLs in HTTP begin with `http://`.

HTTPS (Hypertext Transfer Protocol Secure):
- **Security:** HTTPS is secure. It encrypts data using SSL/TLS (Secure Sockets Layer/Transport Layer Security) protocols, ensuring that data transmitted between the client and server is encrypted and protected from eavesdropping and tampering.
- **Protocol:** It is an extension of HTTP that adds a layer of security by using encryption.
- **Port:** HTTPS uses port 443 by default for communication.
- **URL:** URLs in HTTPS begin with `https://`.

Key Differences:
1. **Encryption:** HTTPS encrypts the data transmitted between the client and server, while HTTP does not provide encryption, making it vulnerable to security threats.
2. **Security:** HTTPS ensures data integrity, authentication, and confidentiality, offering a higher level of security compared to HTTP.
3. **Protocol Extension:** HTTPS is essentially HTTP over a secure SSL/TLS connection, adding a security layer to the standard HTTP protocol.
4. **Port Usage:** HTTP uses port 80, while HTTPS uses port 443 by default.

In summary, the primary difference between HTTP and HTTPS lies in their security measures. HTTPS encrypts data during transmission, providing a secure and encrypted connection, while HTTP transfers data in plain text, lacking the encryption and security features of HTTPS.

#middleware
middleware is a way to process requests globally before they reach the view or after they leave the view
Middleware components are applied in a specific order and can perform various tasks, such as authentication, security, request/response modification, and more.

CommonMiddleware: Adds several helpful headers to HTTP responses and handles URL redirection.
AuthenticationMiddleware: Handles user authentication.
SessionMiddleware: Manages sessions for users.
CsrfViewMiddleware: Adds Cross-Site Request Forgery (CSRF) protection.
SecurityMiddleware: Enforces security best practices, like setting the `X-Content-Type-Options` header.
MessageMiddleware: Enables the use of Django's messaging framework

.

Certainly! Let's break down the process of creating a custom middleware in Django step by step:

Step 1: Create a new Python file for your middleware

Create a new file for your custom middleware. For example, let's name it `custom_middleware.py`. You can place this file in the same directory as your Django app.

```bash
your_project/
|-- your_app/
|   |-- ...
|-- custom_middleware.py
|-- manage.py
|-- your_project/
    |-- ...
```

Step 2: Define your custom middleware class

Open `custom_middleware.py` and define your custom middleware class. In this example, we'll create a middleware that logs information about each incoming request:

```python
# custom_middleware.py

class RequestLoggingMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response

    def __call__(self, request):
        # This code is executed for each request before the view is called.
        self.log_request(request)
        response = self.get_response(request)
        # This code is executed for each response after the view is called.
        return response

    def log_request(self, request):
        # Add your custom logging logic here
        path_info = request.path_info
        method = request.method
        print(f"Request received: {method} {path_info}")
```

Step 3: Configure your middleware in settings.py

Open your `settings.py` file and add the path to your custom middleware class to the `MIDDLEWARE` setting:

```python
# settings.py

MIDDLEWARE = [
    # ...
    'your_app.custom_middleware.RequestLoggingMiddleware',
    # Add other middleware components as needed
]
```

Make sure to replace `'your_app'` with the actual name of your Django app.

Step 4: Restart your Django development server

After making these changes, save the files and restart your Django development server:

```bash
python manage.py runserver
```

Now, your custom middleware should be active. It will log information about each incoming request to the console.

You can customize the `log_request` method in the middleware class to perform any actions you need, such as logging to a file, modifying the request or response, etc.

signals

In Django, signals allow decoupled components to get notified when certain actions occur elsewhere in the application. They provide a way for different parts of a Django application to communicate without being directly tied to each other.

Here's a step-by-step guide on how to use signals in Django:

1. Import the required modules

First, import the necessary modules in your Django app:

```python
# In your app's signals.py file
from django.db.models.signals import Signal
from django.dispatch import receiver
```

2. Create a Signal

Define a signal by instantiating the `Signal` class:

```python
# In your app's signals.py file
my_signal = Signal()
```

3. Create a Receiver Function

Define a function that will be called when the signal is sent. Decorate the function with the `@receiver` decorator, specifying the signal to which it should respond.

```python
# In your app's signals.py file
@receiver(my_signal)
def my_signal_handler(sender, **kwargs):
    # Your custom logic here
    print(f"Signal received from {sender}")
```

4. Connect the Signal

In your app's `apps.py` file, override the `ready` method and connect the signal to the receiver function:

```python
# In your app's apps.py file
from django.apps import AppConfig

class YourAppConfig(AppConfig):
    default_auto_field = 'django.db.models.BigAutoField'
    name = 'your_app'

    def ready(self):
        import your_app.signals  # Import your signals module
```

5. Send the Signal

In any part of your code (e.g., views, models, etc.), when you want to send the signal, use the `send` method:

```python
# In your views, models, or any other module
from django.dispatch import Signal

my_signal.send(sender=None)
```

The `sender` parameter in the `send` method can be any Python object, and it is often set to `None` or the model instance that triggered the signal.

Example Summary

Let's summarize the example:

- **signals.py:**
  ```python
  ```

```
# In your app's signals.py file
from django.db.models.signals import Signal
from django.dispatch import receiver

my_signal = Signal()

@receiver(my_signal)
def my_signal_handler(sender, **kwargs):
    print(f"Signal received from {sender}")
```

- **apps.py:**
```python
# In your app's apps.py file
from django.apps import AppConfig

class YourAppConfig(AppConfig):
    default_auto_field = 'django.db.models.BigAutoField'
    name = 'your_app'

    def ready(self):
        import your_app.signals
```

- **views.py (or any other module):**
```python
# In your views, models, or any other module
from django.dispatch import Signal
from your_app.signals import my_signal

my_signal.send(sender=None)
```

This is a basic example, and signals can be more powerful and flexible depending on the needs of your application. They are commonly used for tasks such as handling post-save signals on models, triggering actions after certain events, and implementing decoupled components in Django projects.

Advanced Filtering with Q Objects:

In Django, `Q` objects provide a way to create complex queries by combining multiple conditions with logical operators (AND, OR). They are particularly useful when you need to construct intricate queries that involve OR conditions, negations, or combinations of both.

Basic Usage:

1. Simple OR condition:**

```
from django.db.models import Q

# Retrieve objects where either field1 equals value1 OR field2 equals value2
result = YourModel.objects.filter(Q(field1=value1) | Q(field2=value2))
```

2. AND condition:

```
# Retrieve objects where field1 equals value1 AND field2 equals value2
result = YourModel.objects.filter(Q(field1=value1) & Q(field2=value2))
```

Advanced Usage:**

3. Combining OR and AND:

```
# Retrieve objects where (field1 equals value1 AND field2 equals value2) OR field3 equals value3
result = YourModel.objects.filter((Q(field1=value1) & Q(field2=value2)) | Q(field3=value3))
```

4. Negation (NOT):

```
# Retrieve objects where NOT (field1 equals value1)
result = YourModel.objects.exclude(Q(field1=value1))
```

5. Complex combinations

```
# Retrieve objects where (field1 equals value1 AND field2 equals value2) OR (field3 equals value3 AND field4 equals value4)
result = YourModel.objects.filter(Q(field1=value1, field2=value2) | Q(field3=value3, field4=value4))
```

Using Q Objects with F Objects:**

`Q` objects can also be combined with `F` objects for more advanced queries:

```
from django.db.models import F

# Retrieve objects where field1 is greater than (field2 plus 5)
result = YourModel.objects.filter(field1__gt=F('field2') + 5)
```

Dynamic Query Building:**

`Q` objects are useful for building dynamic queries based on user inputs or other runtime conditions. You can dynamically combine conditions based on your application's logic.

```
conditions = Q(field1=value1) | Q(field2=value2)

if some_condition:
    conditions &= Q(field3=value3)

result = YourModel.objects.filter(conditions)
```

These examples should provide you with a good starting point for using `Q` objects in Django to perform advanced filtering in your database queries. Keep in mind that you can combine multiple `Q` objects to create complex conditions and make your queries more flexible and powerful.

In Django, field lookups are used to filter querysets based on specific conditions or comparisons.

1. **`__gt` (Greater Than):**

```
# Retrieve objects where the 'field_name' is greater than a certain value
result = YourModel.objects.filter(field_name__gt=value)
```

2. **`__icontains` (Case-Insensitive Contains):**

```
# Retrieve objects where the 'field_name' contains a case-insensitive substring
result = YourModel.objects.filter(field_name__icontains='substring')
```

3. **`__exact` (Exact Match): checks for exact match**
4. **`__iexact` (Case-Insensitive Exact Match):**
5. **`__in` (In a List):**

The `__in` lookup is used to filter objects where the specified field's value is in a given list.

6. **`__startswith` and `__endswith`:**

These lookups are used to filter objects where the specified field's value starts or ends with a given substring.

7. `__range` (Range):

The `__range` lookup is used to filter objects where the specified field's value is within a given range.

10. `__isnull` (Is Null):

The `__isnull` lookup filters objects where the specified field is null or not null.

`__regex` and `__iregex` (Regular Expression Match):

The `__regex` and `__iregex` lookups filter objects based on regular expression matching.

`__gt`, `__lt`, `__gte`, `__lte` (Comparison):

___it = less than, __gte = greater than and equal, __lte = less than and equal
These lookups are used for numerical and date-based comparisons.

`__day`, `__month`, `__year`, etc. (Date-related lookups):

Using aggregates and annotations:

Aggregates:

Aggregates perform calculations on a set of records, returning a single value. Common aggregate functions include `Count`, `Sum`, `Avg`, `Min`, *and* `Max`.

```
from django.db.models import Count,Avg

# Count the number of records
result = YourModel.objects.all().aggregate(record_count=Count('id'))
print(result)  # {'record_count': 42}
```

Annotations:

Annotations add additional fields to each record in a queryset based on aggregated values. They are useful when you want to include aggregate information alongside individual records.

Example 1: Annotate each record with the count of related records.

```
from django.db.models import Count

# Annotate each record with the count of related records
result = YourModel.objects.annotate(related_count=Count('related_model'))
for record in result:
    print(record.related_count)
```

QuerySet is a collection of database queries that allows you to retrieve, filter, and manipulate data from a database table. It acts as an abstraction layer between your application code and the database, providing a high-level, Pythonic interface for database interactions.

Caching

Caching in Django involves storing and retrieving data in a temporary storage to reduce the response time of certain operations. Django provides a caching framework that allows you to use various cache backends. Here's how you can handle caching in Django:

1. Configure Caching in Django Settings:

```
# settings.py

CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.locmem.LocMemCache',
        'LOCATION': 'unique-snowflake',  # Unique identifier for this cache instance
        'TIMEOUT': 300,  # Cache timeout in seconds (5 minutes in this example)
    }
}
```

2. Using the Cache in Views:

Cache a View:

To cache the result of a view, you can use the `cache_page` decorator:

```
from django.views.decorators.cache import cache_page

@cache_page(60 * 15)  # Cache for 15 minutes
def my_cached_view(request):
    # Your view logic here
```

Cache a Function Result:

You can also cache the result of a specific function using the `cache` decorator:

```
from django.core.cache import cache

@cache.cached(timeout=60 * 15)  # Cache for 15 minutes
def my_cached_function():
    # Your function logic here
```

6. Django Middleware Caching:

Django provides cache middleware that can automatically cache entire views or specific pieces of content.

```
# settings.py

MIDDLEWARE = [
    # ...
    'django.middleware.cache.UpdateCacheMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.cache.FetchFromCacheMiddleware',
    # ...
]
```

7. Django Cache Template Tag:

Django also provides a `{% cache %}` template tag to cache portions of your templates.

```
{% load cache %}

{% cache 500 sidebar %}
    <!-- Your sidebar content here -->
{% endcache %}
```

1. Local Memory Cache:

Backend: `'django.core.cache.backends.locmem.LocMemCache'`

This backend stores the cache data in local memory. It's useful for development and testing but not suitable for production when running multiple server instances.

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.locmem.LocMemCache',
        'LOCATION': 'unique-snowflake',
    }
}
```

2. File-Based Cache:

Backend: `'django.core.cache.backends.filebased.FileBasedCache'`

This backend stores the cache data in a directory on the filesystem. It's suitable for single-server deployments.

3. Database Cache:

Backend: `'django.core.cache.backends.db.DatabaseCache'`

This backend stores the cache data in a database table. It's suitable for multi-server deployments where all servers can access the same database.

4. Memcached Cache:

Backend: `'django.core.cache.backends.memcached.MemcachedCache'`

This backend uses Memcached, a distributed memory caching system, to store cache data. It's suitable for multi-server deployments and provides high-performance caching.