1. What is Django?

Answer:
Django is a high-level Python web framework that encourages rapid development and clean, pragmatic design. It follows the "Don't Repeat Yourself" (DRY) principle and the Model-View-Template (MVT) architectural pattern. Django is designed to help developers take applications from concept to completion as quickly as possible by providing a set of ready-to-use components and a focus on automating repetitive tasks.

---

2. Explain the MVT architecture in Django.

Answer:
The MVT (Model-View-Template) architecture in Django consists of three main components:

- **Model:** Represents the data or database layer. It defines the structure of stored data and is usually a Python class.
- **View:** Handles the business logic and interacts with the model to pass data to the template. The view processes HTTP requests, fetches data from the model, and sends it to the template.
- **Template:** Deals with the presentation layer. It defines the HTML structure and displays data passed from the view.

Django uses views to manage the logic of the application, and templates to present the data to the user. Models are used to interact with the database.

---

3. What are Django Models?

Answer:
Django Models are Python classes that represent the structure of the database tables. Each model maps to a single database table and is used to define fields and behaviors of the data you are storing. Models are defined in the models.py file of a Django application.

Example of a Django model:

```
from django.db import models

class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.CharField(max_length=50)
    published_date = models.DateField()
    isbn = models.CharField(max_length=13)
```

---

4. How does Django handle database migrations?

Answer:
Django uses a migration system to handle changes in the database schema. When you make changes to your models (such as adding a field, deleting a field, or altering a field), you create a migration file that captures the changes. You then apply this migration to the database using Django's migration commands.

Common commands for handling migrations:

- python manage.py makemigrations: Creates migration files based on the changes detected in the models.
- python manage.py migrate: Applies the migration to the database.

---

5. What is a QuerySet in Django?

Answer:
A QuerySet is a collection of database queries that can be used to retrieve data from the database. It represents a collection of objects from your database and can be filtered, sliced, and ordered to create more specific queries.

Example of using a QuerySet:

```
# Retrieving all books
books = Book.objects.all()
# Filtering books by a specific author
books_by_author = Book.objects.filter(author="J.K. Rowling")
# Retrieving a single object
book = Book.objects.get(id=1)
```

---

6. What are Django Forms?

Answer:
Django Forms are a framework for handling form validation and processing in Django. Forms provide a way to create and handle HTML forms in Python. They are typically used to handle input from users, validate the data, and save it to the database.

Example of a Django form:

```
python
Copy code
from django import forms
class ContactForm(forms.Form):
   name = forms.CharField(max_length=100)
   email = forms.EmailField()
```

```
    message = forms.CharField(widget=forms.Textarea)
```

---

7. How do you implement user authentication in Django?

Answer:
Django provides a built-in user authentication system, which includes user login, logout, password management, and user groups. You can use Django's built-in views and forms to handle authentication, or create custom views and forms as needed.

Basic steps to implement user authentication:

1. Use the built-in User model from django.contrib.auth.models.
2. Use authentication views provided by django.contrib.auth.views for login and logout.
3. Define authentication URLs in your urls.py.

Example URLs:

```
from django.urls import pathfrom django.contrib.auth import views as auth_views

urlpatterns = [
    path('login/', auth_views.LoginView.as_view(), name='login'),
    path('logout/', auth_views.LogoutView.as_view(), name='logout'),
]
```

---

8. What is middleware in Django?

Answer:
Middleware is a way to process requests globally before they reach the view or after the view has processed them. It is a lightweight plugin that can be used to modify request and response objects. Common uses for middleware include handling sessions, authentication, cross-site request forgery protection, and content compression.

Example of a custom middleware:

```
class SimpleMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response

    def __call__(self, request):
        # Code to execute before the view is called
        response = self.get_response(request)
```

```
    # Code to execute after the view is called
    return response
```

---

9. How does Django's template system work?

Answer:
Django's template system allows developers to create HTML templates with dynamic content. Templates are HTML files with placeholders for dynamic content and logic. The template system uses the Django Template Language (DTL) to render variables, loops, conditionals, and filters.

Example template code:

```
<!DOCTYPE html>
<html>
<head>
  <title>{{ title }}</title>
</head>
<body>
  <h1>{{ message }}</h1>
  {% if user.is_authenticated %}
    <p>Welcome, {{ user.username }}!</p>
  {% else %}
    <p>Please log in.</p>
  {% endif %}
</body>
</html>
```

In views, you pass context to templates:

```
from django.shortcuts import render

def home(request):
  context = {'title': 'Home', 'message': 'Hello, World!'}
  return render(request, 'home.html', context)
```

10. What are signals in Django?

Answer:
Signals are a way for decoupled applications to get notified when certain actions occur elsewhere in the application. Django provides a "signal dispatcher" that allows certain senders to notify a set of receivers when certain actions have taken place.

Commonly used signals include:

- post_save: Sent after a model's save() method is called.
- pre_save: Sent before a model's save() method is called.
- post_delete: Sent after a model's delete() method is called.

```
from django.db.models.signals import post_save
from django.dispatch import receiver
from django.contrib.auth.models import User

@receiver(post_save, sender=User)
def notify_user_created(sender, instance, created, **kwargs):
    if created:
        print(f"User {instance.username} has been created.")
```

11. What is Django ORM?

Answer:
Django ORM (Object-Relational Mapping) is an abstraction layer that allows developers to interact with their database using Python code instead of SQL queries. It provides a high-level API to create, retrieve, update, and delete database records, making it easier to work with different databases without writing raw SQL.

Key features of Django ORM:

- Simplifies database access by converting Python objects into SQL queries.
- Supports multiple database backends, including PostgreSQL, MySQL, SQLite, and Oracle.
- Provides query optimization and caching for efficient database operations.

12. How do you optimize database queries in Django?

Answer:
To optimize database queries in Django, consider the following techniques:

> **Use** select_related() **and** prefetch_related()**:** These methods help reduce the number of database queries by fetching related objects in a single query or by prefetching them in batches.
>
> **books = Book.objects.select_related('author').all()**

· **Use** only() **and** defer()**:** These methods allow you to load only the fields you need or defer loading some fields until they are accessed.

· **Use Database Indexes:** Adding indexes to frequently queried fields can improve query performance.

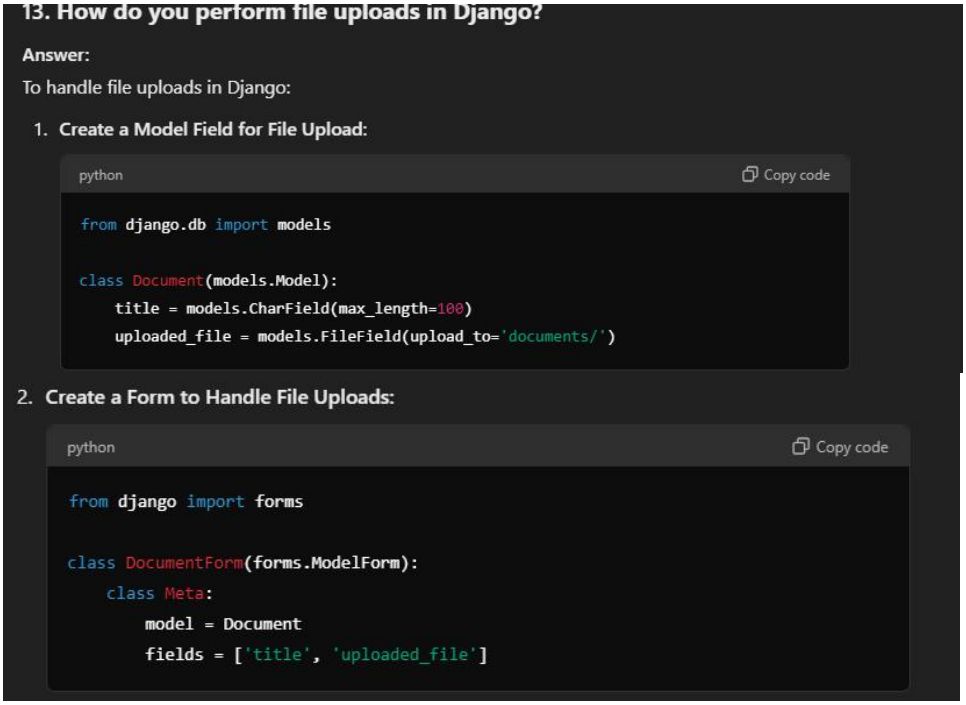
· · **Avoid** N+1 **Query Problem:** Use select_related() and prefetch_related() to avoid making an additional query for each object in a queryset.

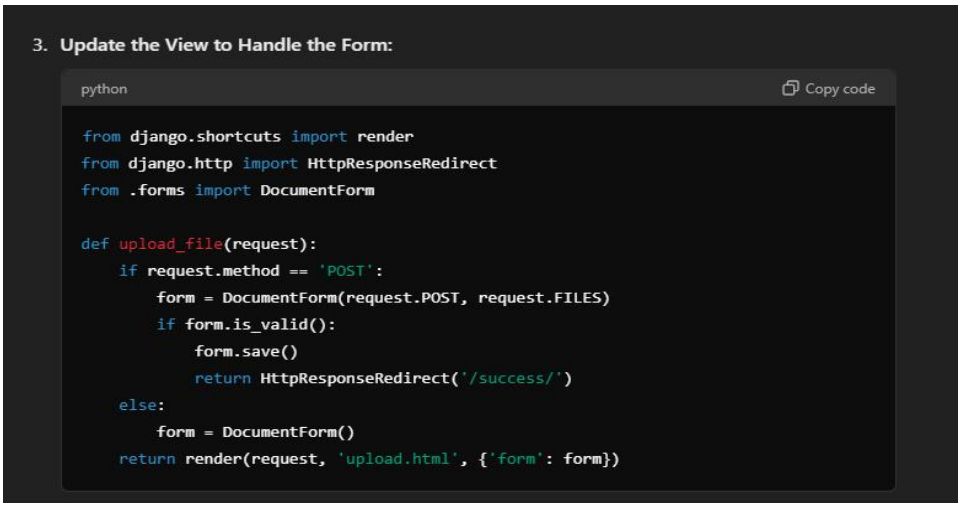
· · **Use Aggregations and Annotations:** Use Django's aggregation functions like Sum, Count, Avg, etc., to perform calculations in the database.

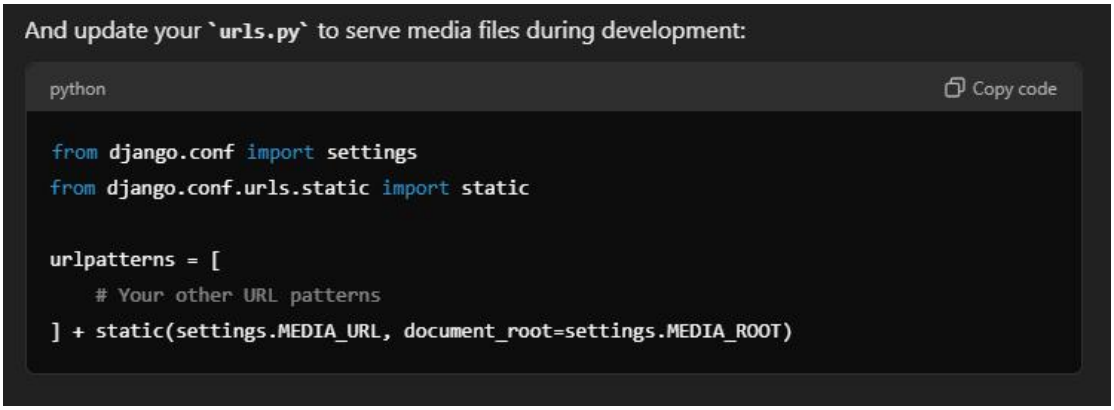
13. How do you perform file uploads in Django?

Answer:
To handle file uploads in Django:

Create a Model Field for File Upload:







14. What is Django Rest Framework (DRF)?

Answer:
Django Rest Framework (DRF) is a powerful and flexible toolkit for building Web APIs in Django. It provides a set of tools and libraries to handle common API tasks, such as serialization, authentication, permissions, and more.

Key features of DRF:

- **Serialization:** Easily convert complex data types, like Django models, into native Python data types for rendering into JSON or XML.
- **Authentication and Permissions:** Built-in support for various authentication methods (Token, OAuth, JWT) and customizable permissions.
- **Browsable API:** Provides an interactive, web-browsable API interface for developers.

```python
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework import status
from .models import Book
from .serializers import BookSerializer

class BookList(APIView):
    def get(self, request):
        books = Book.objects.all()
        serializer = BookSerializer(books, many=True)
        return Response(serializer.data)

    def post(self, request):
        serializer = BookSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data, status=status.HTTP_201_CREATED)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```