Q23. What does MVC stand for in Django context?

MVC stands for **Model–View–Controller**, which is a widely used software architectural pattern for developing web applications.

- **Model**: Represents the **data layer** of the application. It defines the structure of data, handles database queries, and manages data-related logic. In Django, this is implemented through **Model classes** in models.py.
- **View**: Contains the **business logic** of the application. It processes user requests, interacts with models, and returns responses (HTML, JSON, etc.). In Django, this is handled in views.py.
- **Controller**: Responsible for **routing user requests** to the correct view. In Django, this role is performed by the **URL dispatcher** in urls.py.

Although Django technically follows the **MVT (Model–View–Template)** pattern, it still closely maps to MVC, where Django's Template layer is similar to the View in MVC, and Django's View is similar to the Controller in MVC.

Q24. Which command applies migrations in Django?

In Django, the command used to apply database migrations is:

```
python manage.py migrate
```

- **Purpose**: The migrate command updates the actual database schema based on the migration files created by makemigrations.
- **Process**:
    1. You first create or modify models in models.py.
    2. Run python manage.py makemigrations to generate migration files.
    3. Finally, run python manage.py migrate to execute those changes on the database.
- Example: If you add a new field to a model, applying migrations will create that field in the database without losing existing data.

Q25. What is the role of ModelSerializer in DRF?

The **ModelSerializer** in Django REST Framework is a specialized serializer class that automatically generates serializer fields based on a given Django model.

- It reduces boilerplate code by **inferring fields, validators, and relationships** from the model's definition.
- It supports **both serialization** (converting model instances to JSON/XML) and **deserialization** (converting JSON/XML data into model instances).
- **Example**:

```
class StudentSerializer(serializers.ModelSerializer):
    class Meta:
        model = Student
        fields = '__all__'

```

This automatically maps all fields from the Student model to the serializer.

Q26. Name two built-in authentication classes in DRF.

Django REST Framework includes several built-in authentication classes. Two of the most commonly used are:

1. **BasicAuthentication** – Uses HTTP Basic Authentication, where the username and password are sent with each request (suitable for testing or internal APIs).
2. **SessionAuthentication** – Uses Django's built-in session framework to authenticate users (suitable for web applications where users log in via forms).

Both classes can be configured in the DEFAULT_AUTHENTICATION_CLASSES setting.

Q27. What is the default port for Django's development server?

The default port for Django's development server is **8000**.

- When you run the command:

```
python manage.py runserver
```

Django starts a development server accessible at:

http://127.0.0.1:8000

- You can specify a different port by adding it after the command, e.g.:

```
python manage.py runserver
```

which would run the server on port 8080 instead of 8000.

Q28. What is the use of Django signals? Give one example.

Django signals are a feature that allows **different parts of an application to communicate with each other** when certain actions occur. They are used to **trigger specific functions automatically** in response to particular events, without tightly coupling different parts of the code.

- **Purpose**:
    - Decouples event handling logic from the core functionality.
    - Allows automatic execution of code when a specific event happens, such as **saving a model instance** or **user login**.
- **Common built-in signals**:
    - pre_save – Triggered before a model is saved.
    - post_save – Triggered after a model is saved.
    - pre_delete and post_delete – Triggered before/after deletion of a model instance.

Example – Creating a profile automatically when a user is registered:

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

@receiver(post_save, sender=User)
def create_user_profile(sender, instance, created, **kwargs):
    if created:
        Profile.objects.create(user=instance)
```

Here, whenever a **User** instance is created, a corresponding **Profile** instance is automatically generated.

Q29. Define MEDIA_ROOT and MEDIA_URL in Django.

In Django, when working with file uploads (images, documents, etc.), **MEDIA_ROOT** and **MEDIA_URL** are used for managing **media files**.

1. **MEDIA_ROOT**:
    - **Definition**: The **absolute filesystem path** where uploaded media files are stored on the server.
    - **Example**:

```
MEDIA_ROOT = BASE_DIR / 'media'
```

This stores uploaded files in a media folder inside the project directory.

2. **MEDIA_URL**:
    - **Definition**: The **URL path** used to access media files in the browser.
    - **Example**:

```
MEDIA_URL = '/media/'
```

This means uploaded files will be accessible via URLs like /media/filename.jpg.

Usage:

- When a user uploads a file, Django saves it to MEDIA_ROOT and serves it to the user using the MEDIA_URL path.
- You must configure **urlpatterns** to serve media files during development.

Q30. List different types of Permissions in DRF.

In Django REST Framework (DRF), **permissions** are used to control access to API endpoints based on the user's authentication status or role.

Common permission classes include:

1. **AllowAny** – Grants access to any user, authenticated or not.
2. **IsAuthenticated** – Grants access only to authenticated (logged-in) users.
3. **IsAdminUser** – Grants access only to admin or staff users.

4. **IsAuthenticatedOrReadOnly** – Allows read-only access for unauthenticated users, but write access only for authenticated users.
5. **Custom Permissions** – Created by extending BasePermission to implement specific business rules.

Example:

```
from rest_framework.permissions import IsAuthenticated

class MyView(APIView):
    permission_classes = [IsAuthenticated]
```

Q31. What is a Router in DRF?

A **Router** in Django REST Framework is a class that **automatically generates URL patterns** for ViewSets, reducing the need to manually write URL mappings.

Benefits:

- Eliminates repetitive code.
- Automatically supports standard CRUD endpoints.
- Integrates cleanly with DRF's **ViewSet** classes.

Example:

```
from rest_framework.routers import DefaultRouter
from .views import UserViewSet

router = DefaultRouter()
router.register(r'users', UserViewSet)

urlpatterns = router.urls
```

Here, DRF automatically creates routes like:

- /users/ (list & create users)
- /users/{id}/ (retrieve, update, delete user)

Q32. Mention two advantages of using JWT Authentication.

JWT (JSON Web Token) Authentication is a popular method for securing APIs, especially in mobile and single-page applications.

Advantages:

1. **Stateless Authentication** – The server does not store session data; all information needed is in the token itself, making it scalable for large systems.
2. **Cross-Platform Support** – JWTs work well with mobile apps, web apps, and microservices because they are self-contained and language-independent.

Example:
In DRF, JWTs can be implemented using the djangorestframework-simplejwt package, where tokens are sent in the HTTP Authorization header for each request.

Q33. Differentiate between Django Models and Migrations.

Django Models and **Migrations** are both essential components for database management in Django, but they serve different purposes.

Aspect	**Models**	**Migrations**
Definition	Python classes that define the **structure** of database tables.	Scripts that record changes made to models and apply them to the actual database.
Purpose	Represent and manage the application's data layer.	Synchronize the database schema with the current state of the models.
Creation	Written manually in models.py.	Generated automatically using python manage.py makemigrations.
Execution	Do not change the database until migrations are applied.	Applied to the database using python manage.py migrate.
Example	class Student(models.Model): name = models.CharField(max_length=100)	Generated migration file adding the name field to the Student table.

Conclusion: Models define **what the database should look like**, and migrations are the **process of implementing those changes** in the actual database.

Q34. Write steps to customize Django Admin interface.

The **Django Admin** can be customized to make it more user-friendly and tailored to project requirements.

Steps:

1. **Register the Model** in admin.py:

```
from django.contrib import admin
from .models import Student
admin.site.register(Student)
```

2. **Create a Custom ModelAdmin Class**:

```
class StudentAdmin(admin.ModelAdmin):
list_display = ('name', 'email', 'course')
search_fields = ('name',)
list_filter = ('course',)
```

3. **Register Model with Custom Admin Class**:

```
admin.site.register(Student, StudentAdmin)
```

4. **Additional Customizations**:
    - Inline editing for related models.
    - Adding custom form layouts.
    - Grouping fields with fieldsets.

Example: You can display student names, allow searching by email, and filter by course in the admin panel.

Q35. Explain CRUD operations in Django using Forms.

CRUD stands for **Create, Read, Update, Delete**, which are the four basic operations for managing data in an application.

Implementation using Django Forms:

1. **Create** – Use a form to take input and save a new record:

```
if form.is_valid():
    form.save()
```

2. **Read** – Fetch data from the model and display it:

```
students = Student.objects.all()
```

3. **Update** – Pre-fill a form with existing data and save changes:

```
form = StudentForm(instance=student)
```

4. **Delete** – Remove a record from the database:

```
student.delete()
```

Conclusion: Django Forms handle both creation and editing of model instances easily, while model queries handle reading and deleting.

Q36. Describe Permissions in Django REST Framework with an example.

Permissions in DRF determine whether a request should be granted or denied access to a specific resource. They work alongside authentication to **enforce access control**.

Types of Permissions:

- **AllowAny** – Open access.
- **IsAuthenticated** – Only logged-in users.
- **IsAdminUser** – Only admins/staff.
- **IsAuthenticatedOrReadOnly** – Read for all, write for logged-in users.

Example: Restricting a view to authenticated users only:

```
from rest_framework.permissions import IsAuthenticated
from rest_framework.views import APIView

class StudentList(APIView):
    permission_classes = [IsAuthenticated]
```

Conclusion: Permissions ensure that **sensitive API endpoints are secure** and only accessible to authorized users.

Q37. Explain the use of Nested Serializers in DRF.

Nested Serializers are used when one serializer contains another, allowing related model data to be displayed inside the main serializer output.

Purpose:

- To **embed related model data** directly in API responses.
- Useful for one-to-one and one-to-many relationships.

Example:

```
class ProfileSerializer(serializers.ModelSerializer):
    class Meta:
        model = Profile
        fields = ['bio', 'location']

class UserSerializer(serializers.ModelSerializer):
    profile = ProfileSerializer()
    class Meta:
        model = User
        fields = ['username', 'email', 'profile']
```

Output:

```
{
  "username": "nilesh",
  "email": "nilesh@example.com",
  "profile": {
    "bio": "CS Student",
    "location": "Ahmedabad"
  }
}
```

Conclusion: Nested serializers simplify the process of including **related data** in API responses without making multiple API calls.

Q38. Explain Django's MVT architecture with a diagram.

Django follows the **MVT (Model–View–Template)** architecture, which is a variation of MVC. It separates the application into three interconnected layers.

1. Model

- Represents the **data layer**.
- Defines the structure of the database using Python classes in models.py.

- Handles queries, relationships, and database operations.

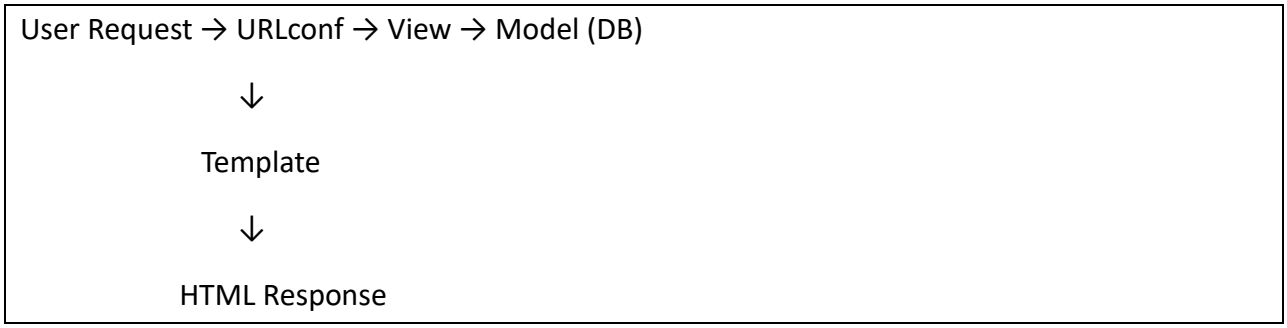
2. View

- Contains the **business logic**.
- Receives HTTP requests, processes data via models, and sends back responses.
- Written in views.py.

3. Template

- The **presentation layer**.
- HTML files containing Django Template Language (DTL) for rendering dynamic content.

Flow of MVT:



Example: When a user requests /students/,

1. URLconf maps it to student_list view.
2. View fetches data from Student model.
3. Template displays it as HTML.

Q39. Describe the process of creating and applying migrations in Django.

Migrations are Django's way of propagating changes made to models into the database schema.

Steps:

1. **Create/Modify Models** – Add or update fields in models.py.
2. **Generate Migrations** – Run:

```
python manage.py makemigrations
```

This creates migration files inside the migrations folder.

3. **Apply Migrations** – Run:

```
python manage.py migrate
```

This applies the changes to the database.

4. **Check Status** – Use:

5.

```
python manage.py showmigrations
```

to see applied and pending migrations.

Example: Adding an email field to a Student model requires creating a migration and applying it to update the database schema.

Q40. Explain the steps to implement file upload in Django with media management.

Step 1 – Configure MEDIA_ROOT and MEDIA_URL in settings.py:

```
MEDIA_ROOT = BASE_DIR / 'media'
MEDIA_URL = '/media/'
```

Step 2 – Create a Model with FileField or ImageField:

python

CopyEdit

```
class Document(models.Model):
    file = models.FileField(upload_to='documents/')
```

Step 3 – Create an HTML Form with enctype="multipart/form-data":

```
<form method="post" enctype="multipart/form-data">
    {% csrf_token %}
    <input type="file" name="file">
    <button type="submit">Upload</button>
```

```
</form>
```

Step 4 – Handle file in the View:

```
def upload_file(request):
    if request.method == 'POST':
        form = DocumentForm(request.POST, request.FILES)
        if form.is_valid():
            form.save()
```

Step 5 – Serve Media Files in Development:

```
urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

Q41. Discuss Django's authentication system for login and logout functionality.

Django provides a **built-in authentication system** via django.contrib.auth.

Login Process:

1. **Authenticate user** – Use authenticate() with username and password.
2. **Log in user** – Use login(request, user) to start a session.
3. **Restrict access** – Use @login_required decorator on views.

Logout Process:

1. Use logout(request) to clear the session data.
2. Redirect to the homepage or login page.

Example:

```
from django.contrib.auth import authenticate, login, logout

def user_login(request):
    user = authenticate(username='nilesh', password='1234')
    if user:
        login(request, user)

def user_logout(request):
    logout(request)
```

Q42. Explain Django REST Framework's ViewSets and Routers with an example.

ViewSets in DRF combine logic for handling multiple actions (list, create, retrieve, update, delete) into a single class.

Routers automatically generate URL patterns for ViewSets, avoiding manual URL mapping.

Example:

python

CopyEdit

```
class StudentViewSet(viewsets.ModelViewSet):
    queryset = Student.objects.all()
    serializer_class = StudentSerializer

router = DefaultRouter()
router.register(r'students', StudentViewSet)
urlpatterns = router.urls
```

Advantages:

- Less boilerplate code.
- Automatic route generation.

Q43. Describe JWT Authentication using djangorestframework-simplejwt with setup steps.

Step 1 – Install:

bash

CopyEdit

```
pip install djangorestframework-simplejwt
```

Step 2 – Update settings.py:

python

CopyEdit

```
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'rest_framework_simplejwt.authentication.JWTAuthentication',
    ),
}
```

Step 3 – Add Token Endpoints in urls.py:

python

CopyEdit

```
from rest_framework_simplejwt.views import TokenObtainPairView, TokenRefreshView

urlpatterns = [
    path('token/', TokenObtainPairView.as_view()),
    path('token/refresh/', TokenRefreshView.as_view()),
]
```

Step 4 – Authenticate Requests:
Send token in header:

makefile

CopyEdit

```
Authorization: Bearer <access_token>
```

Q44. Explain the role of Django templates in the MVT architecture with an example.

Templates are the **presentation layer** in Django's MVT architecture. They define **how data is displayed** to the user using HTML and Django Template Language (DTL).

Features:

- Variable interpolation: {{ variable }}
- Tags for logic: {% if %}, {% for %}
- Template inheritance with {% extends %}

Example:

html

CopyEdit

```
<h1>Welcome {{ user.username }}</h1>
{% for student in students %}
    <p>{{ student.name }}</p>
{% endfor %}
```

Q45. Describe the process of creating a custom user model in Django.

Step 1 – Inherit from AbstractUser or AbstractBaseUser:

python

CopyEdit

```
class CustomUser(AbstractUser):
    phone_number = models.CharField(max_length=15)
```

Step 2 – Update settings.py:

python

CopyEdit

```
AUTH_USER_MODEL = 'myapp.CustomUser'
```

Step 3 – Create and run migrations.

Benefits:

- Add extra fields like phone number, profile picture.
- Modify authentication logic.

Q46. Explain how to handle file validation in Django forms.

Method: Override the clean_<fieldname>() method in the form to validate file type, size, or name.

Example:

python

CopyEdit

```
def clean_file(self):
    file = self.cleaned_data['file']
    if file.size > 2*1024*1024:
        raise forms.ValidationError("File too large")
    return file
```

Q47. Write token-based authentication vs session-based authentication in Django REST Framework.

Token-Based	Session-Based
Stateless	Stateful
Stores token in client	Stores session ID in server
Better for APIs & mobile	Better for web apps

Token-Based	Session-Based
No server storage needed	Requires database/server storage

Q48. Write the steps to implement API versioning in Django REST Framework.

Step 1 – Configure DEFAULT_VERSIONING_CLASS in settings:

python

CopyEdit

```
REST_FRAMEWORK = {
    'DEFAULT_VERSIONING_CLASS': 'rest_framework.versioning.URLPathVersioning'
}
```

Step 2 – Update urls.py with version paths:

python

CopyEdit

```
path('v1/students/', StudentViewSet.as_view({'get': 'list'})),
path('v2/students/', StudentViewSetV2.as_view({'get': 'list'})),
```

Step 3 – Maintain separate views/serializers for each version.