Django

Django Framework

Definition
Django is a high-level Python web framework used to build secure, scalable, and fast web applications.

- It follows the **MVT (Model–View–Template)** architecture.

Official website: Django Software Foundation
Framework: Django

Why Django Is Important?
For Freshers & Interviews:

- Used to build full-stack web applications
- Secure by default
- Built-in admin panel
- Handles authentication
- Used in real-world companies

Companies using Django include:

- Instagram
- Pinterest
- Mozilla

MVT Architecture (Very Important)
Django follows:

1. **Model**
   Handles database (tables, fields, relationships)
2. **View**
   Handles business logic
3. **Template**
   Handles frontend (HTML)

Flow:
User → URL → View → Model → Template → Response

Features of Django

- Built-in ORM (Object Relational Mapping)
- Admin Panel
- Authentication System
- Security (CSRF, SQL injection protection)
- URL Routing
- Middleware Support
- REST API support (with Django REST Framework)

Installing Django

```
pip install django
```

Check version:

```
django-admin --version
```

Creating a Django Project

```
django-admin startproject myproject
```

Structure:

```
myproject/
    manage.py
```

```
    myproject/
        settings.py
        urls.py
        wsgi.py
```

Creating an App

```
python manage.py startapp myapp
App Structure:
myapp/
    models.py
    views.py
    admin.py
    urls.py
```

Simple Example

Model (models.py)

```
from django.db import models

class Student(models.Model):
    name = models.CharField(max_length=100)
    age = models.IntegerField()
```

View (views.py)

```
from django.http import HttpResponse

def home(request):
    return HttpResponse("Hello Django")
```

URL (urls.py)

```
from django.urls import path
from . import views

urlpatterns = [
    path('', views.home),
]
```

Running the Server

```
python manage.py runserver
Open:
http://127.0.0.1:8000/
```

Django ORM (Interview Important)

```
Instead of writing SQL:
Student.objects.create(name="John", age=20)
Django automatically converts it into SQL.
```

Django Admin Panel

```
Create superuser:
python manage.py createsuperuser
Access:
/admin
You get automatic admin dashboard.
```

Advantages of Django

- Fast development
- Secure
- Scalable
- Clean architecture
- Large community support

Django vs Flask (Interview Question)

Feature	Django	Flask
Type	Full-stack	Micro-framework
Admin panel	Built-in	Not built-in
ORM	Built-in	Optional
Best for	Large apps	Small apps

Framework comparison: Flask

Real-Time Project Structure Example

```
revpay/
   manage.py
   users/
   payments/
   transactions/
   templates/
```

Used in enterprise-level applications.

Common Interview Questions

Q1: What is Django?
A: High-level Python web framework.
Q2: What architecture does Django follow?
A: MVT (Model-View-Template)
Q3: What is ORM?
A: Object Relational Mapping.
Q4: How to create project?
A: django-admin startproject projectname
Q5: What is manage.py?
A: Command-line utility for Django project.

Quick Revision Summary

- Django = Python web framework
- Follows MVT architecture
- Has built-in admin panel
- Uses ORM
- Secure and scalable

Django Installation Guide

Framework: Django
Maintained by: Django Software Foundation

Prerequisites

Before installing Django, you need:

Python Installed
Check version:
python --version
Recommended: Python 3.8+
If not installed, download from:
Python Software Foundation

Check pip
pip --version
If not installed:
python -m ensurepip --upgrade

(Recommended) Create Virtual Environment
Why?
To isolate project dependencies.
Create Virtual Environment
python -m venv venv
Activate It
Windows:
venv\Scripts\activate
Mac/Linux:
source venv/bin/activate
After activation, you will see (venv) in terminal.

Install Django
pip install django

Verify Installation
django-admin --version
If version number appears → Installation successful.

Create a Django Project
django-admin startproject myproject
Move inside project:
cd myproject
Run server:
python manage.py runserver
Open browser:
http://127.0.0.1:8000/
You should see the Django welcome page.

Upgrade Django (Optional)
pip install --upgrade django

Install Specific Version (Interview Question)
pip install django==4.2

Common Errors & Fixes
django-admin not recognized
Use:
python -m django --version
ModuleNotFoundError
Activate virtual environment properly.

Best Practice for IT Projects

- Always use virtual environment
- Freeze dependencies

```
pip freeze > requirements.txt
```

Install from file:

```
pip install -r requirements.txt
```

Quick Installation Summary

1. Install Python
2. Create virtual environment
3. Activate it
4. Run pip install django
5. Verify using django-admin --version
6. Create project using startproject

Django Project Structure

Framework: Django
When you create a project using:

```
django-admin startproject myproject
```

You get the following structure

Default Project Structure

```
myproject/
│
├── manage.py
│
└── myproject/
    ├── __init__.py
    ├── settings.py
    ├── urls.py
    ├── asgi.py
    └── wsgi.py
```

File-by-File Explanation (Very Important for Interviews)

---

1. manage.py

Command-line utility for interacting with the project.
Used for:

```
python manage.py runserver
python manage.py startapp appname
python manage.py migrate
```

Think of it as the project controller.

---

2. init.py

- Makes the folder a Python package
- Usually empty
- Required for module recognition

---

3. settings.py (Very Important)
Main configuration file.
Contains:
- Installed apps
- Database configuration
- Middleware
- Templates settings
- Static files configuration
- Secret key
- Debug mode

Example:
```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
]
```
Most commonly asked file in interviews.

---

4. urls.py
URL routing configuration.
Maps URL → View
Example:
```
from django.urls import path
from . import views

urlpatterns = [
    path('', views.home),
]
```
Controls application navigation.

---

5. wsgi.py
Used for deployment (production servers like Gunicorn).
WSGI = Web Server Gateway Interface.
Used when deploying Django app.

---

6. asgi.py
Used for asynchronous support.
ASGI = Asynchronous Server Gateway Interface.
Required for:
- WebSockets
- Real-time apps
- Async features

After Creating an App
When you run:
```
python manage.py startapp myapp
```
New structure:
```
myapp/
├── migrations/
├── __init__.py
├── admin.py
├── apps.py
├── models.py
├── tests.py
└── views.py
```

App File Explanation

---

models.py
Used to define database tables.

---

views.py
Contains business logic.

---

admin.py
Register models for admin panel.

---

migrations/
Stores database migration files.

---

tests.py
Used for writing test cases.

---

apps.py
App configuration file.

Real-World Project Structure (Company Level)

```
revpay/
│
├── manage.py
├── users/
├── payments/
├── transactions/
├── templates/
├── static/
└── media/
```

Explanation:

- templates → HTML files
- static → CSS, JS, Images
- media → Uploaded files

This is how enterprise Django applications are structured.

Important Interview Questions

Q1: What is the role of manage.py?
A: Used to execute Django commands.

Q2: What is settings.py?
A: Main configuration file of Django project.

Q3: Difference between project and app?
A: Project = Entire application
App = Single module within project

Q4: What is WSGI?
A: Interface between web server and Django application.

Project vs App (Quick Comparison)

Project	App
Whole application	Small functional module
Contains settings	Contains models, views
Created once	Can create multiple

Quick Revision Summary

- manage.py → command tool
- settings.py → configuration
- urls.py → routing
- wsgi.py → production deployment
- asgi.py → async support
- models.py → database
- views.py → business logic

Django Apps

Framework: Django

What is a Django App?

A **Django App** is a reusable module inside a Django project that handles a specific functionality.

- Project = Entire application
- App = Single functional component

Example:

In a banking system:

- users app
- payments app
- transactions app

Why Django Apps Are Important?

- Modular development
- Reusable components
- Easy maintenance
- Scalable architecture
- Used in real-world enterprise projects

Creating a Django App

Inside project directory:

```
python manage.py startapp myapp
```

App Structure

```
myapp/
├── migrations/
├── __init__.py
├── admin.py
├── apps.py
├── models.py
├── tests.py
└── views.py
```

File-by-File Explanation (Interview Important)

---

1. models.py

Used to define database tables.

Example:

```
from django.db import models
```

```
class Student(models.Model):
    name = models.CharField(max_length=100)
    age = models.IntegerField()
```

---

2. views.py
Contains business logic.
Example:

```
from django.http import HttpResponse

def home(request):
    return HttpResponse("Welcome to Django App")
```

---

3. admin.py
Register models for admin panel.

```
from django.contrib import admin
from .models import Student

admin.site.register(Student)
```

---

4. apps.py
App configuration file.

```
from django.apps import AppConfig

class MyappConfig(AppConfig):
    name = 'myapp'
```

---

5. migrations/
Stores database change files.
Used with:

```
python manage.py makemigrations
python manage.py migrate
```

---

6. tests.py
Used to write test cases.

After Creating App – Important Step
You must register the app in settings.py:

```
INSTALLED_APPS = [
    'myapp',
]
```

Without this → app will not work.

Connecting App to Project
Step 1: Create urls.py inside app

```
from django.urls import path
from . import views
urlpatterns = [
    path('', views.home),
]
```

Step 2: Include in project urls.py

```
from django.urls import path, include
urlpatterns = [
    path('app/', include('myapp.urls')),
]
```

Real-World Example (Company Level)

```
revpay/
│
├── users/
├── payments/
├── notifications/
├── transactions/
```

Each app handles one business module.
This makes the system clean and scalable.

Best Practices for Django Apps

- Keep each app focused on one responsibility
- Avoid putting all logic in one app
- Use separate apps for large features
- Keep models, views organized

Common Interview Questions

Q1: What is the difference between project and app?
A: Project = Entire website
App = Specific functionality module

Q2: Can we create multiple apps in one project?
A: Yes. Django supports multiple apps.

Q3: Why do we register app in settings.py?
A: So Django recognizes it.

Q4: Can an app be reused in another project?
A: Yes. That is the main advantage of Django apps.

Quick Revision Summary

- App = Reusable module
- Created using startapp
- Must register in INSTALLED_APPS
- Contains models, views, admin, migrations
- Multiple apps inside one project

Django Settings Configuration

Framework: Django
The **settings.py** file is the main configuration file of a Django project.
It controls database, apps, middleware, security, static files, templates, and more.

Location

```
myproject/
    manage.py
    myproject/
        settings.py   ← Main configuration file
```

Important Sections in settings.py (Interview Important)

BASE_DIR

```
from pathlib import Path
BASE_DIR = Path(__file__).resolve().parent.parent
```

- Represents project root directory.

Used for defining file paths.

SECRET_KEY

```
SECRET_KEY = 'your-secret-key'
```

- Used for security (sessions, CSRF, encryption)
- Never expose in production
- Should be stored in environment variables

DEBUG

```
DEBUG = True
```

- True → Development mode
- False → Production mode

Always set DEBUG = False in production.

ALLOWED_HOSTS

```
ALLOWED_HOSTS = ['127.0.0.1', 'localhost']
```

Defines which domains can access your app.
In production:

```
ALLOWED_HOSTS = ['yourdomain.com']
```

INSTALLED_APPS (Very Important)

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'myapp',
]
```

- Registers apps
- Without this, app won’t work

MIDDLEWARE

```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
]
```

Middleware handles:

- Security
- Sessions
- Authentication
- Request/Response processing

DATABASES (Very Important)

Default (SQLite):

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR / 'db.sqlite3',
    }
}
```

Production Example (MySQL):

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'dbname',
        'USER': 'root',
        'PASSWORD': 'password',
        'HOST': 'localhost',
        'PORT': '3306',
```

```
    }
}
```

TEMPLATES

```
TEMPLATES = [
    {
        'DIRS': [BASE_DIR / 'templates'],
        'APP_DIRS': True,
    },
]
```

Used for HTML template configuration.

STATIC FILES

```
STATIC_URL = '/static/'
STATICFILES_DIRS = [BASE_DIR / 'static']
```

Used for:

- CSS
- JS
- Images

MEDIA FILES (User Uploads)

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

Used for uploaded files.

Custom Settings Example

You can create custom variables:

```
EMAIL_HOST = 'smtp.gmail.com'
```

Then access inside project:

```
from django.conf import settings
print(settings.EMAIL_HOST)
```

Production Best Practices (Very Important)

- Use environment variables for SECRET_KEY
- Set DEBUG = False
- Configure ALLOWED_HOSTS
- Use PostgreSQL/MySQL instead of SQLite
- Separate dev and prod settings

Example structure:

```
settings/
    base.py
    dev.py
    prod.py
```

Common Interview Questions

Q1: What is settings.py?
A: Main configuration file of Django project.

Q2: What happens if app not added to INSTALLED_APPS?
A: Django will not recognize it.

Q3: What is DEBUG?
A: Development mode flag.

Q4: What is ALLOWED_HOSTS?
A: List of allowed domain names.

Q5: Where do we configure database?
A: Inside DATABASES dictionary.

Quick Revision Summary

- settings.py controls entire project configuration
- Important sections:
  - SECRET_KEY
  - DEBUG
  - INSTALLED_APPS
  - DATABASES
  - MIDDLEWARE
  - TEMPLATES
  - STATIC & MEDIA
- Must configure properly for production

Django Models & Django ORM

Framework: Django

What is a Django Model?

A **Model** defines the structure of your database tables using Python classes.

- Each model = One database table
- Each field = One table column

Example Model

```
from django.db import models

class Student(models.Model):
    name = models.CharField(max_length=100)
    age = models.IntegerField()
    email = models.EmailField(unique=True)

    def __str__(self):
        return self.name
```

This creates a table like:

| id | name | age | email |

Important Model Fields (Interview Important)

Field Type	Description
CharField	Text (requires max_length)
TextField	Large text
IntegerField	Integer number
FloatField	Decimal numbers
BooleanField	True/False
DateField	Date
DateTimeField	Date & Time
EmailField	Email validation
FileField	File upload
ImageField	Image upload
ForeignKey	Relationship (Many-to-One)
OneToOneField	One-to-One
ManyToManyField	Many-to-Many

Model Meta Options

```
class Meta:
    db_table = "student_table"
    ordering = ['name']
```

Used to customize table behavior.

What is Django ORM?

ORM = Object Relational Mapping

It allows you to interact with database using Python code instead of writing SQL.

Example:

Instead of writing:

```
SELECT * FROM student;
```

You write:

```
Student.objects.all()
```

Django automatically converts it to SQL.

ORM CRUD Operations

Create

```
Student.objects.create(name="John", age=20, email="john@gmail.com")
```

Or:

```
s = Student(name="Mike", age=22)
s.save()
```

Read

Get all records:

```
Student.objects.all()
```

Filter:

```
Student.objects.filter(age=20)
```

Get single record:

```
Student.objects.get(id=1)
```

Update

```
s = Student.objects.get(id=1)
s.age = 25
s.save()
```

Or:

```
Student.objects.filter(id=1).update(age=30)
```

Delete

```
s = Student.objects.get(id=1)
s.delete()
```

Or:

```
Student.objects.filter(age=20).delete()
```

Migrations (Very Important)

After creating model:

```
python manage.py makemigrations
python manage.py migrate
```

- makemigrations → Creates migration file
- migrate → Applies changes to database

Relationships in Models

- **ForeignKey (Many-to-One)**

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

class Student(models.Model):
    name = models.CharField(max_length=100)
    course = models.ForeignKey(Course, on_delete=models.CASCADE)
```
One course → Many students

- **OneToOneField**

```
class Profile(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE)
```
One user → One profile

- **ManyToManyField**

```
class Student(models.Model):
    name = models.CharField(max_length=100)
    courses = models.ManyToManyField(Course)
```
Many students ↔ Many courses

QuerySet Methods (Interview Important)

Method	Purpose
all()	Get all records
filter()	Filter records
get()	Single record
exclude()	Exclude records
order_by()	Sorting
count()	Count records
exists()	Check existence
first()	First record
last()	Last record

Advanced ORM Example

Filter with condition:

```
Student.objects.filter(age__gt=18)
```

Operators:

Lookup	Meaning
__gt	Greater than
__lt	Less than
__gte	Greater or equal
__lte	Less or equal
__icontains	Case-insensitive contains

Example:

```
Student.objects.filter(name__icontains="john")
```

Real-Time Example (Company Level)

In banking system:

- User model
- Transaction model
- Payment model

All related using ForeignKey and ManyToMany relationships.
ORM handles all database logic without writing SQL.

Common Interview Questions
Q1: What is Django ORM?
A: Tool to interact with database using Python objects.
Q2: What is QuerySet?
A: Collection of database queries.
Q3: Difference between filter() and get()?
- filter() → Returns multiple records
- get() → Returns single record

Q4: What is makemigrations?
A: Creates migration file for model changes.
Q5: What is on_delete=models.CASCADE?
A: Deletes related records automatically.

Best Practices
- Always run migrations after model changes
- Use filter() instead of get() when unsure
- Avoid writing raw SQL unless necessary
- Use related_name for better relationship access

Quick Revision Summary
- Model = Database table
- ORM = Python way to interact with DB
- CRUD via objects
- Relationships: FK, OneToOne, ManyToMany
- Use migrations to apply changes

Django Views and Templates

Framework: Django
Django follows **MVT (Model–View–Template)** architecture.
- **Model** → Database
- **View** → Business logic
- **Template** → Frontend (HTML)

Django Views
What is a View?
A **View** is a Python function (or class) that handles HTTP requests and returns HTTP responses.
- It contains business logic.

Function-Based View (FBV)
Example:
```
# views.py
from django.http import HttpResponse

def home(request):
    return HttpResponse("Hello Django")
```
Connect to URL:
```
# urls.py
```

```
from django.urls import path
from . import views

urlpatterns = [
    path('', views.home),
]
```

Returning HTML Template from View

Instead of plain text:

```
from django.shortcuts import render

def home(request):
    return render(request, 'home.html')
```

Django Templates

What is a Template?

A **Template** is an HTML file used to display data dynamically.

Location:

```
project/
   templates/
       home.html
```

Configure Templates in settings.py

```
TEMPLATES = [
    {
        'DIRS': [BASE_DIR / 'templates'],
        'APP_DIRS': True,
    },
]
```

Passing Data to Template

View:

```
def home(request):
    context = {
        "name": "John",
        "age": 25
    }
    return render(request, 'home.html', context)
```

Template (home.html):

```
<h1>Hello {{ name }}</h1>
<p>Age: {{ age }}</p>
```

Output:

```
Hello John
Age: 25
```

- {{ }} is used to display variables.

Template Tags (Interview Important)

If Condition

```
{% if age > 18 %}
    Adult
{% else %}
    Minor
```

```
{% endif %}
```

For Loop

```
{% for student in students %}
    <p>{{ student.name }}</p>
{% endfor %}
```

URL Tag

```
<a href="{% url 'home' %}">Home</a>
```

Static Files in Templates

In settings.py:

```
STATIC_URL = '/static/'
```

In template:

```
{% load static %}
<link rel="stylesheet" href="{% static 'css/style.css' %}">
```

Template Inheritance (Very Important)

Used for common layout.

---

base.html

```
<!DOCTYPE html>
<html>
<head>
    <title>{% block title %}My Site{% endblock %}</title>
</head>
<body>
    {% block content %}
    {% endblock %}
</body>
</html>
```

---

home.html

```
{% extends 'base.html' %}

{% block title %}Home Page{% endblock %}

{% block content %}
<h1>Welcome</h1>
{% endblock %}
```

- Avoids repeating header/footer code.

Class-Based Views (CBV)

Example:

```
from django.views import View
from django.http import HttpResponse

class HomeView(View):
    def get(self, request):
        return HttpResponse("Hello from Class View")
```

Views Types (Interview Question)

Type	Description
Function-Based	Simple, easy

Type	Description
Class-Based	More powerful, reusable

Real-Time Flow
User → URL → View → Fetch Data from Model → Send to Template → Display in Browser
This is complete request lifecycle.

Common Interview Questions
Q1: What is a View?
Handles request and returns response.

Q2: What is render()?
Shortcut function to return template with context.

Q3: What is context?
Dictionary used to pass data to template.

Q4: What is template inheritance?
Reusing common layout using extends.

Q5: Difference between FBV and CBV?
FBV → Simple functions
CBV → Class-based, reusable

Best Practices
- Keep business logic in views
- Keep HTML only in templates
- Use template inheritance
- Use class-based views for large projects

Quick Revision Summary
- View → Handles request
- Template → Displays data
- render() → Connects view and template
- Use {{ }} for variables
- Use {% %} for logic
- Template inheritance avoids duplication

URL Configuration & Django Admin

PART-1 – URL Configuration (Routing)

What is URL Configuration?
URL configuration (urls.py) maps **URL patterns → Views.**
- It decides which view runs when a user visits a specific URL.

Project Level urls.py
Location:
```
myproject/
    urls.py
```
Example:

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('app/', include('myapp.urls')),
]
```

Explanation:

- admin/ → Django Admin
- include() → Connect app-level URLs

App Level urls.py

Create inside app:

```
# myapp/urls.py
from django.urls import path
from . import views

urlpatterns = [
    path('', views.home, name='home'),
]
```

URL Pattern Syntax

```
path('route/', view_function, name='url_name')
```

Example:

```
path('about/', views.about, name='about')
```

Dynamic URL Parameters (Interview Important)

Example:

```
path('student/<int:id>/', views.student_detail)
```

View:

```
def student_detail(request, id):
    return HttpResponse(f"Student ID: {id}")
```

Types of Converters

Converter	Type
str	String
int	Integer
slug	Slug text
uuid	UUID
path	Full path

Using URL Name in Template

```
<a href="{% url 'home' %}">Home</a>
```

Best practice: always use name.

URL Flow (Important)

User → URL → urls.py → View → Response

PART-2 — Django Admin

Django has a built-in admin panel.

What is Django Admin?
An automatic backend interface for managing database records.
No need to build admin manually.

Create Superuser
python manage.py createsuperuser
Enter:

- Username
- Email
- Password

Run Server
python manage.py runserver
Visit:
http://127.0.0.1:8000/admin/
Login with superuser credentials.

Register Model in Admin
Inside admin.py:

```
from django.contrib import admin
from .models import Student

admin.site.register(Student)
```

Now model appears in admin dashboard.

Customizing Admin (Interview Important)
Example:

```
from django.contrib import admin
from .models import Student

class StudentAdmin(admin.ModelAdmin):
    list_display = ('name', 'age', 'email')
    search_fields = ('name',)
    list_filter = ('age',)
admin.site.register(Student, StudentAdmin)
```

Features:

- list_display → Show fields in table
- search_fields → Search option
- list_filter → Filter sidebar

Admin Customization Options

Option	Purpose
list_display	Columns shown
list_filter	Sidebar filters
search_fields	Search box
ordering	Default sorting
readonly_fields	Make fields read-only

Why Django Admin Is Powerful?

- Automatic CRUD interface
- No frontend coding needed
- Secure authentication
- Easy model management

URL vs Admin (Interview Difference)

URL Configuration	Django Admin
Routes user requests	Backend dashboard
Connects views	Manages database
Defined in urls.py	Built-in feature

Common Interview Questions

Q1: What is include() in URLs?
A: Used to connect app-level URLs.
Q2: What is dynamic URL?
A: URL that accepts parameters.
Q3: How to register model in admin?
A: Using admin.site.register(ModelName)
Q4: What is name parameter in path()?
A: Used for reverse URL lookup.
Q5: Where is admin URL defined?
A: In project-level urls.py

```
path('admin/', admin.site.urls)
```

Quick Revision Summary

- urls.py maps URL → View
- Use include() for app routing
- Dynamic URLs use converters
- Django Admin is built-in backend
- Register models in admin.py
- Customize admin using ModelAdmin

Django REST framework

Django REST framework (DRF) is a powerful toolkit built on top of Django to create RESTful APIs quickly and securely.

Why Use DRF?

- Easily build REST APIs
- Automatic JSON serialization
- Authentication & permissions support
- Browsable API interface
- Built-in validation
- Supports CRUD operations

Installation

```
pip install djangorestframework
Add to settings.py:
INSTALLED_APPS = [
    ...
    'rest_framework',
]
```

Basic DRF Components

Model

```
# models.py
from django.db import models

class Student(models.Model):
    name = models.CharField(max_length=100)
    age = models.IntegerField()
```

Serializer

Converts model data → JSON and JSON → model data

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

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

View

```
# views.py
from rest_framework.decorators import api_view
from rest_framework.response import Response
from .models import Student
from .serializers import StudentSerializer

@api_view(['GET'])
def student_list(request):
    students = Student.objects.all()
    serializer = StudentSerializer(students, many=True)
    return Response(serializer.data)
```

URL Configuration

```
# urls.py
from django.urls import path
from .views import student_list

urlpatterns = [
    path('students/', student_list),
]
```

HTTP Methods Supported

- GET → Retrieve data
- POST → Create data
- PUT → Update full data
- PATCH → Update partial data
- DELETE → Delete data

Class-Based Views (Better Practice)

```
from rest_framework.views import APIView

class StudentList(APIView):
    def get(self, request):
        students = Student.objects.all()
```

```
        serializer = StudentSerializer(students, many=True)
        return Response(serializer.data)
```

ViewSets (Most Powerful)

```
from rest_framework.viewsets import ModelViewSet

class StudentViewSet(ModelViewSet):
    queryset = Student.objects.all()
    serializer_class = StudentSerializer
In urls.py:
from rest_framework.routers import DefaultRouter
router = DefaultRouter()
router.register('students', StudentViewSet)

urlpatterns = router.urls
```

Authentication Options

- Basic Authentication
- Session Authentication
- Token Authentication
- JWT Authentication

Permissions

```
from rest_framework.permissions import IsAuthenticated

permission_classes = [IsAuthenticated]
```

Advantages of DRF

- Fast API development
- Clean code structure
- Scalable
- Built-in validation
- Easy integration with frontend
