1) What is a web server? (example: role of nginx)

A **web server** is software that accepts network requests (usually HTTP/HTTPS), serves files, and forwards dynamic requests to application code. It runs on a machine with an IP and listens on ports (80 for HTTP, 443 for HTTPS). Typical responsibilities:

- **Serve static files** (HTML, CSS, JS, images) efficiently.
- **Terminate TLS** (handle HTTPS handshakes) or pass the encrypted traffic further.
- **Reverse proxy**: receive incoming requests and forward (proxy) them to one or more backend application servers (e.g., Gunicorn, uWSGI, Daphne).
- **Load balancing** across multiple backend servers.
- **Caching** responses to reduce load and latency.
- **Request filtering, rate limiting, compression, HTTP header management**.

nginx specifically is a high-performance web server and reverse proxy optimized for many concurrent connections. In a typical Django deployment nginx handles TLS, serves static files, and proxies dynamic requests to the Django process manager (e.g., Gunicorn).

---

2) How do a client (browser) and server (backend) communicate?

At a high level: the browser and server communicate over the network using TCP/IP and the HTTP protocol (often HTTP over TLS = HTTPS). Main steps:

1. **DNS lookup**: browser resolves the hostname to an IP address.
2. **Transport layer (TCP)**: the browser opens a TCP connection to the server IP and port (three-way handshake).
3. **(If HTTPS) TLS handshake**: negotiate encryption keys, verify certificates.
4. **HTTP request**: the browser sends an HTTP request (method like GET/POST, URL path, headers, optional body).
5. **Server processes** request and sends an **HTTP response** (status code, headers, body).
6. **Connection management**: connection may be kept open (HTTP keep-alive, HTTP/2 multiplexing) for further requests.
7. **Browser renders** response, may make more requests (images, CSS, XHR/fetch).

Communication uses **requests/responses**, HTTP headers (Content-Type, Cookie, Cache-Control, Authorization, etc.), and may include cookies, tokens, or session identifiers for state.

---

3) What is HTTP? What happens when you visit a URL?

HTTP (HyperText Transfer Protocol) is an application-layer protocol for transferring hypermedia (HTML, JSON, files) between clients and servers. It defines methods (GET, POST, PUT, DELETE...), status codes (200, 301, 404, 500...), headers, and a message structure (start line → headers → body).

Visiting a URL (high-level sequence):

1. **Browser parses the URL** (scheme, host, port, path, query). Example: https://example.com:443/path?x=1.
2. **DNS resolution**: resolve example.com → IP.
3. **Open connection**: TCP handshake to IP:port.
4. **TLS handshake** (if https) to establish secure channel.
5. **Send HTTP request**: typically GET /path?x=1 HTTP/1.1 + headers (Host, User-Agent, Accept, Cookie...).
6. **Server responds**: status code + headers + body (HTML).
7. **Browser processes response**: parses HTML, builds DOM, fetches additional resources (CSS/JS/images). Each resource may trigger new HTTP requests (often over the same connection with HTTP/2).
8. **Render & execute**: apply CSS, run JS, handle XHR/fetch calls.

Behind the scenes you also get redirects (3xx), caching behavior, cookies and authentication, and potential errors (4xx/5xx).

---

4) Where does a Django app fit in the network architecture?

Django is the **application layer / web framework** that implements your site's business logic and dynamic responses. Typical deployment stack:

Browser ⟶ (nginx or load balancer) ⟶ **WSGI/ASGI server** (Gunicorn/uWSGI/Daphne) ⟶ **Django application** ⟶ **Database** / cache / other services.

5) Flow: Browser → DNS → Network → Web Server → Django App → Database — what happens at each step?

Browser → The user enters a URL in the browser.

DNS → The domain name is translated into an IP address.

Network → An internet connection to the server is established (TCP handshake).

Web Server → Nginx receives the request.

Django App → The request is processed by the application logic.

Database → If data is needed, Django reads from or writes to the database.

The response travels back from the database → Django → Web Server → Browser.

---

6. What is Django's architecture?

Django is a **web framework** that helps build websites and web applications. Its architecture is designed to separate different parts of the app for easier management, maintenance, and development.

Django uses a design pattern similar to **MVC** but slightly different, called **MVT (Model-View-Template)**.

MVC (Model-View-Controller) vs. MVT (Model-View-Template)

- **MVC** is a classic software design pattern.
    - **Model:** Represents the data and business logic.
    - **View:** Displays data to the user (UI).
    - **Controller:** Handles user input and interacts between Model and View.
- **MVT** is Django's variation:
    - **Model:** Same as MVC, it handles data, database structure, and business rules.
    - **View:** In Django, a view is actually the **controller** part; it processes user requests, gets data from the Model, and returns a response.
    - **Template:** This is the presentation layer — HTML files that display the data to the user.

In short:

MVC Component	Django Equivalent
Model	Model
View	Template
Controller	View (in Django)

Roles of Core Components in Django

- **URL**
  The URL is like the app's address map. It maps URLs (web addresses) to the correct view function/class that should handle the request. For example, if you visit /home, the URL config decides which view handles this.
- **View**
  The view is the business logic layer. It receives the user request, interacts with the model to get or update data, and then decides which template to render or what data to return (like JSON in APIs).
- **Model**
  The model defines the data structure — it's how Django represents database tables as Python classes. It handles data validation, database queries, and relationships.
- **Template**
  The template is the HTML (or other output formats) that presents data to the user. It can include placeholders or variables that the view fills with actual data before sending the final page.

7. What is a Python virtual environment (virtualenv)?

A Python virtual environment is a **self-contained folder** that includes its own Python interpreter and set of installed libraries, separate from the system-wide Python installation.

Why is it important in Django or any Python project?

- **Isolates dependencies:** Different projects may require different versions of the same Python packages. Virtual environments keep these separate so they don't conflict.

- **Avoids polluting the system:** You don't install packages globally, which helps keep your system clean.
- **Reproducibility:** You can recreate the exact environment for your project on another machine using requirements.txt, ensuring the project behaves the same way.
- **Easy management:** You can upgrade or uninstall packages for one project without affecting others.

In Django projects, where you often use many external packages (like Django REST Framework, Pillow, etc.), virtualenv ensures all required packages are managed per project.

---

8. What is the difference between Django REST Framework and Django Templates?

Django Templates

What are Django Templates?

- Django Templates are a system for **generating HTML pages** dynamically.
- Templates contain **HTML code** mixed with **template language syntax** — like variables, loops, and conditional statements — that lets you insert data dynamically.
- When a Django **view** passes data to a template, the template fills in the placeholders with actual data and generates a complete HTML page to send back to the user's browser.

How do Django Templates work?

1. The user makes a request (e.g., visits a webpage).
2. Django's URL dispatcher calls a **view** function or class.
3. The view fetches any required data from the database (via Models).
4. The view passes the data to a **template**.
5. The template processes the data and generates an HTML page.
6. Django sends the rendered HTML back to the user's browser.

Features of Django Templates:

- Supports variables, filters, and tags to manipulate data.
- Helps separate **logic** (in views/models) from **presentation** (in templates).

- Easy to create dynamic pages with user data or other information.
- Can include other templates for reusability (like headers, footers).
- Can use **template inheritance** for DRY (Don't Repeat Yourself) design.

When to use Django Templates?

- When building **traditional multi-page websites**.
- When you want the server to render full HTML pages.
- When SEO (search engine optimization) matters because the content is fully rendered on the server.
- When you have limited JavaScript or want simple server-side rendered pages.

Django REST Framework (DRF)

What is Django REST Framework?

- DRF is a powerful toolkit built on top of Django to create **RESTful APIs**.
- An API (Application Programming Interface) allows different programs or services to communicate, typically exchanging data like JSON.
- DRF makes it easy to build APIs that provide data to frontend apps (React, Angular, Vue), mobile apps, or other services.

How does DRF work?

1. The user (or client app) sends an HTTP request (GET, POST, PUT, DELETE) to an API endpoint.
2. DRF maps this request to a **viewset** or **API view**.
3. The view interacts with the **model** to get or update data.
4. The data is converted (serialized) into a format like **JSON** or **XML**.
5. The serialized data is sent back as an HTTP response.

Features of DRF:

- Built-in support for **serialization** — converting Django models to JSON and vice versa.
- Handles different HTTP methods to create, retrieve, update, or delete data.

- Supports **authentication and permissions** (e.g., token-based, OAuth).
- Provides **browsable API** for easy testing.
- Supports filtering, pagination, and versioning of APIs.

When to use Django REST Framework?

- When building **single-page applications (SPA)** or mobile apps that consume data from an API.
- When your frontend is separated from the backend (e.g., React or Vue frontend + Django backend).
- When you want to provide a **public API** for your data or services.
- When you want more control over data format and interactions than just HTML pages.

---

9. How can we create a simple Django project?

Step 1: Install Django (if you haven't yet)

bash

CopyEdit

pip install django

Step 2: Create a new Django project

Use the command:

bash

CopyEdit

django-admin startproject myproject

This creates a folder named myproject with this basic structure:

markdown

CopyEdit

myproject/

  manage.py

  myproject/

__init__.py

settings.py

urls.py

asgi.py

wsgi.py

- **manage.py**: A command-line utility to interact with your project (runserver, migrate, etc.).
- **myproject/** (inner folder): Contains project settings and configuration.
    - settings.py: Configuration file for your project (database, apps, middleware, etc.).
    - urls.py: Main URL routing for your project.
    - wsgi.py & asgi.py: For deployment configurations.

Step 3: Run the development server

Navigate into the project folder and start the server:

bash

CopyEdit

cd myproject

python manage.py runserver

Now, opening http://127.0.0.1:8000/ in your browser will show the default Django welcome page.

---

10. How can we create an app inside a Django project?

Step 1: Create an app

Inside your project folder (where manage.py is), run:

bash

CopyEdit

python manage.py startapp myapp

This creates a folder myapp/ with this structure:

markdown

CopyEdit

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

- **models.py**: Define database models (tables) here.
- **views.py**: Define what data to show and which templates to use.
- **admin.py**: Configure the admin interface for this app.
- **migrations/**: Database migration files.
- **apps.py**: App configuration.

Step 2: Connect the app to the project

In myproject/settings.py, add 'myapp' to the INSTALLED_APPS list:

python

CopyEdit

```
INSTALLED_APPS = [
    # default Django apps...
    'myapp',
]
```

This tells Django to include your app when it runs.

Write a short explanation of what happens when visiting a URL (answering the questions), including:

When you visit a URL in a **Django** app:

1. **Network** – Your browser sends an **HTTP request** over the internet to the server's IP address (via DNS lookup). The server (and possibly a web server like Nginx) receives the request.
2. **Request Path** – The part of the URL after the domain (e.g., /home/) is extracted from the request.
3. **URL Dispatcher** – Django's urls.py checks this path against defined URL patterns and decides which **view function** should handle it.
4. **View Handling** – The matched view function runs. It may query the database, perform calculations, or prepare data for the response.
5. **Template Rendering** – If the view returns HTML, it often uses a **template** (.html file with placeholders) that Django fills with dynamic data before sending the complete HTML back to the browser.