🧠 Introduction to Django and Web Application Architecture

🌍 What is Web Development?

Web development is about **building websites or web applications** that users interact with through a browser.
It usually involves **two main sides**:

- **Frontend (Client side)** → What the user sees (HTML, CSS, JavaScript)
- **Backend (Server side)** → Handles logic, data storage, and processing (Python, Django, Node.js, etc.)

When both frontend and backend are handled using Python, we call it **Python Full Stack Development**.

🏗️ Three-Tier Architecture in Web Applications

Django (like most modern frameworks) follows a **3-tier architecture** — this ensures modularity and clean separation of concerns.

1. ### Presentation Layer (Frontend/UI)
    - What the user sees and interacts with.
    - Made with HTML, CSS, JavaScript.
    - Examples: Buttons, forms, tables, etc.
    - **In Django**, this is usually handled by *templates* (HTML files) or through frontend frameworks (React, Angular, etc.) consuming APIs.
2. ### Application Layer (Logic Layer)

        - The **heart** of the application.
        - Handles logic, validations, request processing.
        - In Django, this includes:
            - Views (business logic)
            - Models (data structures)
            - Forms, Serializers, and Middleware

3. **Data Layer**
    - Responsible for interacting with the **database**.
    - Django provides ORM (Object-Relational Mapper) — we write Python code instead of SQL queries.
    - Supported databases: SQLite, MySQL, PostgreSQL, Oracle, etc.

🔗 APIs — The Bridge Between Systems

API (Application Programming Interface) is a **communication link between two software systems**.
It defines how one system can talk to another using structured requests and responses.

For example:

- Your frontend React app calls Django API → Django fetches data from the database → sends back a JSON response.

🔄 Request-Response Life Cycle

1. **Client Sends Request** (browser, Postman, frontend app)
2. **Server Receives Request**
3. **Server Validates & Processes Data**

4. **Server Fetches Data** (from DB or logic)

5. **Server Sends Response** (JSON, HTML, etc.)

All this communication is handled through **APIs**.

👨‍💻 Python Full Stack Developer Path

To become a **Python Full Stack Developer**, you need to master both the **frontend** and **backend** sides of the web.

🧩 Libraries vs Frameworks

Concept	Description	Example
Library	Collection of modules or functions for specific tasks	NumPy (math), Pandas (data), Requests (HTTP)
Framework	A structured collection of libraries and conventions to build complete applications	Django, Flask, FastAPI

🔹 In simple terms:
Libraries help you do a task,
Frameworks tell you how to organize your entire application.

🏗️ Popular Python Web Frameworks

Framework	Use Case	Description
Flask	Small to medium apps	Lightweight, minimalistic, flexible
Django	Medium to large apps	Full-featured, built-in ORM, admin, authentication
Django REST Framework (DRF)	API-only applications	Extension of Django for building RESTful APIs

---

⚙️ Django REST Framework (DRF)

- **Open-source library built on top of Django**
- Used exclusively for creating REST APIs.
- **REST** = *Representational State Transfer* → an architecture that defines rules for designing APIs.
- Uses standard HTTP methods:
    - `GET` → Fetch data
    - `POST` → Create data
    - `PUT/PATCH` → Update data
    - `DELETE` → Delete data

🧠 Types of APIs

1. **REST APIs** → Most popular, lightweight, uses JSON (Django REST Framework)
2. **SOAP APIs** → XML-based, older, used in enterprise systems
3. **GraphQL** → More modern, allows flexible querying (Facebook introduced it)

In our class, we'll learn **REST APIs** because they are the industry standard today.

✅ Advantages of REST APIs

- Platform independent (works on web, mobile, etc.)
- Lightweight (uses JSON)

- Easy to integrate and debug
- Scalable and modular
- Follows standard HTTP conventions

⚙️ Setting Up Django

1️⃣ Create a Virtual Environment

A **virtual environment** is an **isolated workspace** for each project.
It allows you to install packages specific to that project without affecting global Python settings.

Types of environments:

- **Global environment:** Same packages for all projects.
- **Local environment:** Each project has its own dependencies.

Command:

```
python -m venv my_env
```

2️⃣ Activate the Virtual Environment

Windows (CMD):

```
my_env\Scripts\activate
```

PowerShell (if error): `Set-ExecutionPolicy -Scope CurrentUser -ExecutionPolicy RemoteSigned`

Mac/Linux:

```
source my_env/bin/activate
```

3️⃣ Install Django & DRF

```
pip install django djangorestframework
```

```
python -m django --version
```

📁 Django Project Structure

When you create a new project:

```
django-admin startproject project_name
```

You'll get a folder structure like:

```
project_name/
    manage.py
    project_name/
        __init__.py
        settings.py
        urls.py
        asgi.py
        wsgi.py
```

Explanation:

- **`__init__.py`** → Makes this directory a Python package.
- **`settings.py`** → Contains configurations (database, installed apps, middleware, etc.)
- **`urls.py`** → Routes requests to different parts of the app.
- **`asgi.py`** → Asynchronous server gateway (for async web servers).
- **`wsgi.py`** → Web server gateway (for traditional web servers).
- **`manage.py`** → Command-line utility to interact with Django.

🚀 Running the Development Server

```
python manage.py runserver
```

You’ll see something like:

```
Starting development server at http://127.0.0.1:8000/
```

Now open your browser and visit http://127.0.0.1:8000.

⚡ Default Server Ports

Technology	Default Port
HTML (Live Server)	5500
Node.js (React/Next.js)	3000
MySQL	3306
Django	8000

🧩 Adding Apps to a Django Project

A **project** can contain multiple **apps** — each handling a specific function.

Example:

```
project/
    blog_app/
    user_app/
    product_app/
```

Command to create an app:

```
python manage.py startapp app_name
```

Then register your app inside **`settings.py`** under `INSTALLED_APPS`.