**Learning Notes:** [**Data Engineering Workshop 2**](https://github.com/internsujank/DataEngineering-Workshop2)

## **Django Basics**

### **What is Django?**

* Django is a Python framework that helps in creating websites easily.
* It has ready-made features like login system, database connection, and CRUD (Create, Read, Update, Delete).
* Focuses on reusing code so we don’t write everything from scratch.

### **How Django Works?**

Django follows the **MVT Pattern (Model – View – Template):**

* **Model (M)** → Connects to the database and provides data.
* **View (V)** → Decides what data to send and which template to use.
* **Template (T)** → The HTML page (with Django tags) that shows the final output.

### **Model**

* Handles the database.
* Uses **ORM (Object Relational Mapping)** → lets us work with databases using Python instead of writing SQL queries.
* Usually written in **models.py**.

**Example:** A "Student" model to store names, age, etc.

### **View**

* A function or method that takes a user request and returns a response.
* Uses data from models and sends it to templates.
* Written in **views.py**.

### **Template**

* A file that describes how the output should look.
* Uses HTML + Django Tags (like {{ variable }}).
* Stored in the **templates folder**.

**Example:**

<h1>My Homepage</h1>

<p>My name is {{ firstname }}.</p>

### **URLs**

* Connects a web address (URL) to a view function.
* Defined in **urls.py**.

**Example:** If user visits /home, Django decides which view to show.

### **Generic Workflow**

1. User requests a URL.
2. Django checks **urls.py** and sends request to the correct view.
3. The view may use data from models.
4. The view sends data to a template.
5. Template returns the final HTML page to the browser.

## Project & App Setup

### **1. Install Django**

****python -m pip install Django

Installs Django framework.

django-admin --version

Checks installed Django version.

### **2. Create Django Project**

****django-admin startproject myworld

Creates a new project named **myworld** with default project files.

### **3. Run Development Server**

****python manage.py runserver

Starts Django development server at http://127.0.0.1:8000/.

Stop server → CTRL + C.

### **4. Create Django App**

****python manage.py startapp members

Creates a new app named **members** inside the project.

### **5. Add Views (basic response)**

Views are Python functions that return a response.

Example: show “Hello World” when user visits the app URL.

### **6. Configure URLs**

* **App URLs**: Define routes for the app (members/urls.py).
* **Project URLs**: Include app routes in myworld/urls.py.  
  Example: Visiting /members/ will load the **members app**.

### **7. Templates**

****mkdir members/templates



Creates a folder for HTML files.

Templates define webpage design and structure.

### **8. Register App in Settings**

Add app name (members) inside INSTALLED\_APPS in **settings.py** so Django recognizes it.

### **9. Database Migrations**

****python manage.py migrate

Applies database changes and creates default tables.

python manage.py makemigrations members

Creates migration file for new models (tables).

python manage.py migrate

Executes migration and creates actual table in database.

### **10. Django Admin Setup**

****python manage.py createsuperuser

Creates an admin user for accessing Django Admin site.

python manage.py runserver

Run the server and visit http://127.0.0.1:8000/admin/ to log in and manage database

(CRUD).

## **Dockerizing Django Project**

### **1. Create Dockerfile**

****mkdir dockerfiles

Defines how Django project runs inside a Docker container.

Basic Dockerfile includes:

* Python base image
* Install Django
* Copy project files
* Set working directory

### **2. Create docker-compose.yml**

****vi docker-compose.yml

Defines services (containers).

* web\_service → Django app container
* Maps port **8000:8000**
* Mounts project folder inside container

### **3. Run Containers**

****docker-compose up -d

Starts containers in detached mode.

docker exec -it workshop\_web\_container sh

Access the Django container shell.

python manage.py runserver 0:8000

Runs Django server inside container. Access: http://127.0.0.1:8000/.

## **Adding PostgreSQL Service**

### **4. Update docker-compose.yml**

Add PostgreSQL service (psql-db):

* Image: postgres:14
* User: postgres
* Password: 123456
* Expose port 5432
* Attach volume to persist data

Bring containers up:

docker-compose up -d

Now two containers run:

* psql-db → PostgreSQL database
* workshop\_web\_container → Django app

### **5. Create Database in Postgres**

****docker exec -it psql-db sh

Creates a new database named **member\_db**.

## **Connect Django to Postgres**

### **6. Update Dockerfile (Django container)**

* Install PostgreSQL dependencies and psycopg2 package.
* Rebuild containers:

docker-compose up --build -d

### **7. Update settings.py (Database Config)**

Replace SQLite with Postgres:

ENGINE: 'django.db.backends.postgresql'

### **8. Run Migrations**

****docker exec -it workshop\_web\_container sh

Applies Django models to Postgres database.

### **9. Verify Database Tables**

Inside Postgres container:

docker exec -it psql-db sh

Lists tables → should see **members\_members** and other Django tables.

### **10. Create Superuser & Run Server**

****python manage.py createsuperuser

Access Django Admin at:

http://localhost:8000/admin

**Django REST API**

## **What is an API?**

* **API (Application Programming Interface):** A set of code that enables communication between two software applications.
* Example:
  + To show Google Maps or Twitter feeds on your site, you use their **APIs**, not their internal code.

## **What is REST API?**

* **REST (Representational State Transfer):** Architectural style for designing networked applications.
* REST APIs allow **simple, flexible, and stateless** communication (usually over HTTP).

## **Django REST Framework (DRF)**

* Built on top of Django to create web APIs.
* Handles:
  + Serialization (convert Django models ↔ JSON/XML).
  + CRUD operations easily.
* Uses Django ORM for database interaction.

## **Getting Started with DRF**

### **Step 1: Create Model (Students)**

* Create a model with fields: first\_name, last\_name, address, roll\_number, mobile, branch.
* Register model in **admin.py** for visibility.

### **Step 2: Apply Migrations**

Commands (inside container workshop\_web\_container):

docker exec -it workshop\_web\_container sh

## **Adding Student Data**

* Go to**http://0.0.0.0:8000/admin/**
* Login → Add entries to the **Students** model.
* Add at least **10 student records**.

## **Step 3: Create API Views**

* Use **class-based views** (View class).
* get() → fetch student data (by roll number or branch).
* post() → add new student record.
* Responses returned as **JSON**.

## **Step 4: Setup Endpoints**

In urls.py, define endpoints:

* /rest/student/<rollno> → Get student by roll number.
* /rest/student/<branch> → Get students by branch.
* /rest/student/ → Add new student (POST).

## **Step 5: Test API with cURL**

### **GET Requests**

**Get student by roll number:**

curl -X GET http://0.0.0.0:8000/members/rest/student/1



**Get students by branch:**

curl -X GET http://0.0.0.0:8000/members/rest/student/CA



### **POST Requests**

**Add new student**

curl -X POST http://0.0.0.0:8000/members/rest/student/ \

-d "first\_name=Shamith&last\_name=H&address=MK&roll\_number=3&mobile=897654354&branch=CA”



**Unit Tests**

## **What is Unit Testing?**

* **Unit Testing:** Testing small, independent parts of software (units) one by one.
* Purpose: To ensure each part works as expected before combining them.

## **Example (Step-wise Testing)**

Project:

1. Scrape data from Python blogs.
2. Save data into the database.
3. Display data on Django web app.

Testing plan:

* Test scraping logic (does it fetch data?).
* Test DB storage (is data saved correctly?).
* Test web display (is data shown properly?).

## **Advantages of Unit Testing**

* Developer checks if each part works correctly.
* Easier to find and fix bugs early.
* Focused testing → small chunks are easier to debug.

## **Disadvantages of Unit Testing**

* Time-consuming (each unit tested separately).
* May miss bugs caused by integration between parts.
* Need to **retest after every change**.

## **Implementing Unit Tests**

### **Run Tests (inside Docker container)**

****python3 -m unittest unittest\_api.py

This script runs tests like:

* test\_get\_student\_branch
* test\_get\_all\_students\_for\_branch

### **View Test Results**

****cat event\_logs.txt

## **Expected Results**

**Status Code 200** → Success (API worked).

Example:

self.assertEqual(result.status\_code, 200)

Ensure response is not empty:

self.assertNotEqual(result, None)

## **Python unittest Module**

* Built-in Python module for writing automated tests.
* Tests are written in classes that inherit from unittest.TestCase.
* Common assertions:
  + assertEqual(a, b) → Check if values match.
  + assertNotEqual(a, b) → Check if values don’t match