Fitness Studio Booking API

Name: PavanSaiKumar Kakumanu

Experience: 1+ Year Python Developer Project Duration: 12/06/2025 – 15/06/2025

2025

**Project Overview**

This project is a backend API built using **FastAPI** to simulate a booking system for a fictional fitness studio. It allows users to:

* View upcoming classes
* Book a spot in a class
* Retrieve their bookings via email  
  The API handles slot availability, input validation, and time zone conversion.

**Tools and Technologies Used**

| **Component** | **Tool/Service** |
| --- | --- |
| Language | Python 3.11 |
| Framework | FastAPI |
| Database | PostgreSQL (hosted on AWS RDS) |
| Timezone Handling | pytz, python-dateutil |
| Server | Uvicorn |
| Dependency Mgmt | requirements.txt |
| API Testing | Postman, cURL |
| Containerization | Docker |

**Project Folder Structure**

**fitness\_booking**/

├── main.py # Main FastAPI application with API routes

├── models.py # SQLModel classes for FitnessClass and Booking

├── crud.py # Business logic: fetch classes, create bookings

├── database.py # PostgreSQL (AWS RDS) database connection setup

├── utils.py # Timezone conversion and utility functions

├── Dockerfile # Docker containerization setup

├── requirements.txt # Python dependencies

├── .env # Environment variables (e.g., DATABASE\_URL)

├── static/ # Static assets (if any)

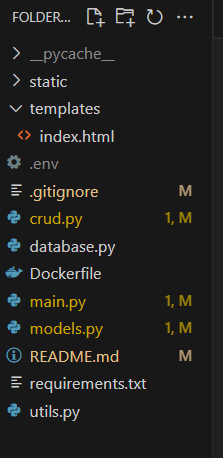
├── templates/

│ └── index.html # HTML template (for optional frontend view)

├── .gitignore # Git ignored files

├── README.md # Project overview and usage instructions

└── \_\_pycache\_\_/ # Auto-generated Python bytecode (ignored in Git)



**Step-by-Step Development Process**

Implementing API Endpoints

I developed all the required API endpoints for the fitness booking system using FastAPI, following RESTful principles. These endpoints are implemented in main.py, with the core logic modularized across the crud.py, models.py, database.py, and utils.py files.

**Endpoint: GET /classes**

* Purpose: List all upcoming fitness classes (e.g., Yoga, Zumba).
* Process:
  + Queries all classes from the PostgreSQL database.
  + Converts datetime to the user’s timezone.
  + Returns class name, instructor, IST datetime, and available slots.

**Endpoint: POST /book**

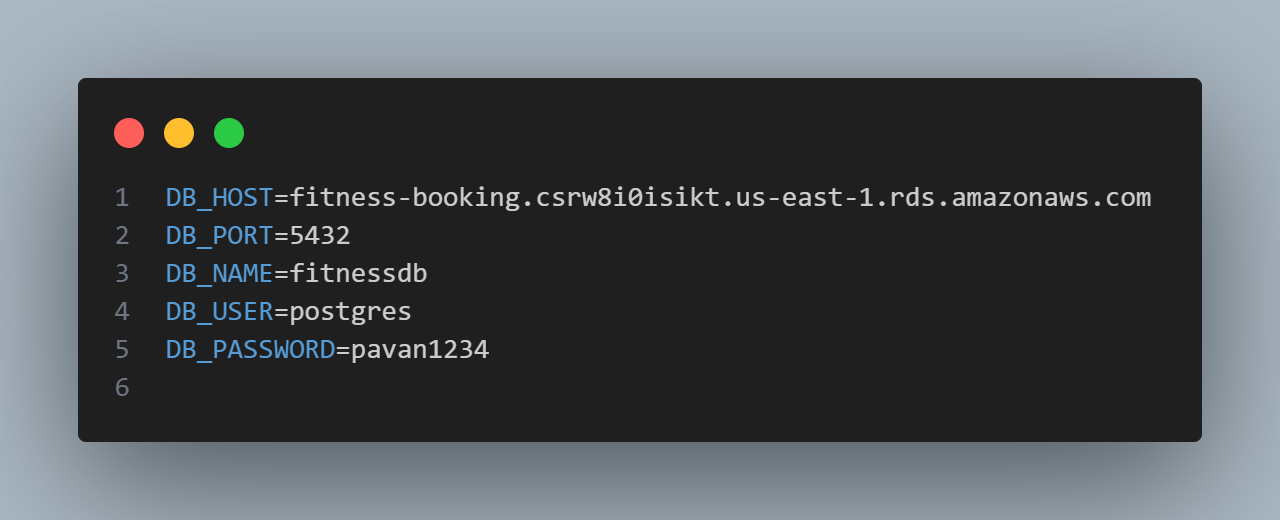
* Purpose: Book a spot in a fitness class.
* Input: class\_id, client\_name, client\_email
* Process:
  + Checks if the class exists and has available slots.
  + Creates a booking entry in the PostgreSQL database.
  + Decrements the available slot count by 1.
* Error Handling:
  + If class not found or no slots left, returns a 400 error(with a clear message to user no slots are available).

**Endpoint: GET /bookings?email=...**

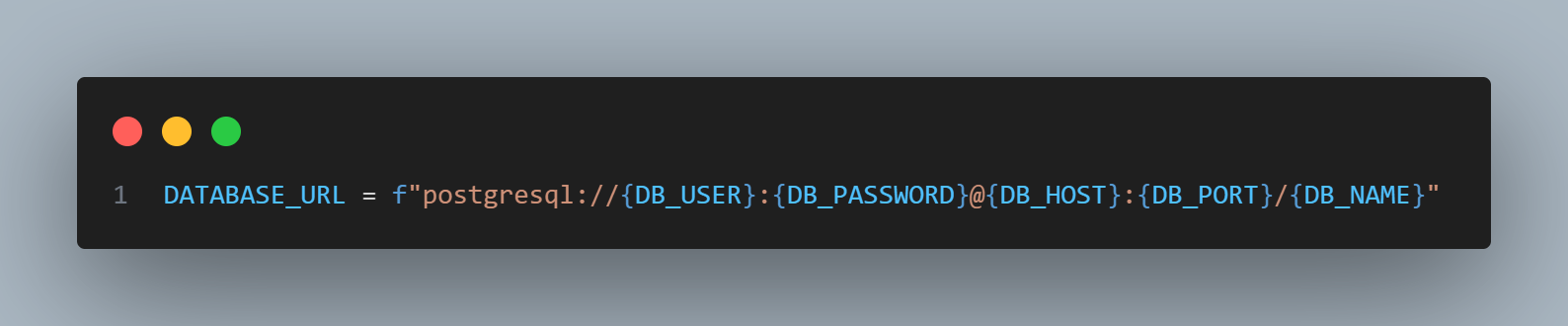
* Purpose: Fetch all bookings by a user using their email address.
* Process:
  + Filters bookings in the database by the given email.
  + Returns list of class names and booking times.

**Database: AWS RDS PostgreSQL**

* Used a hosted PostgreSQL database on **Amazon RDS**.
* Connection configured via DATABASE\_URL in environment variables (loaded using python-dotenv).

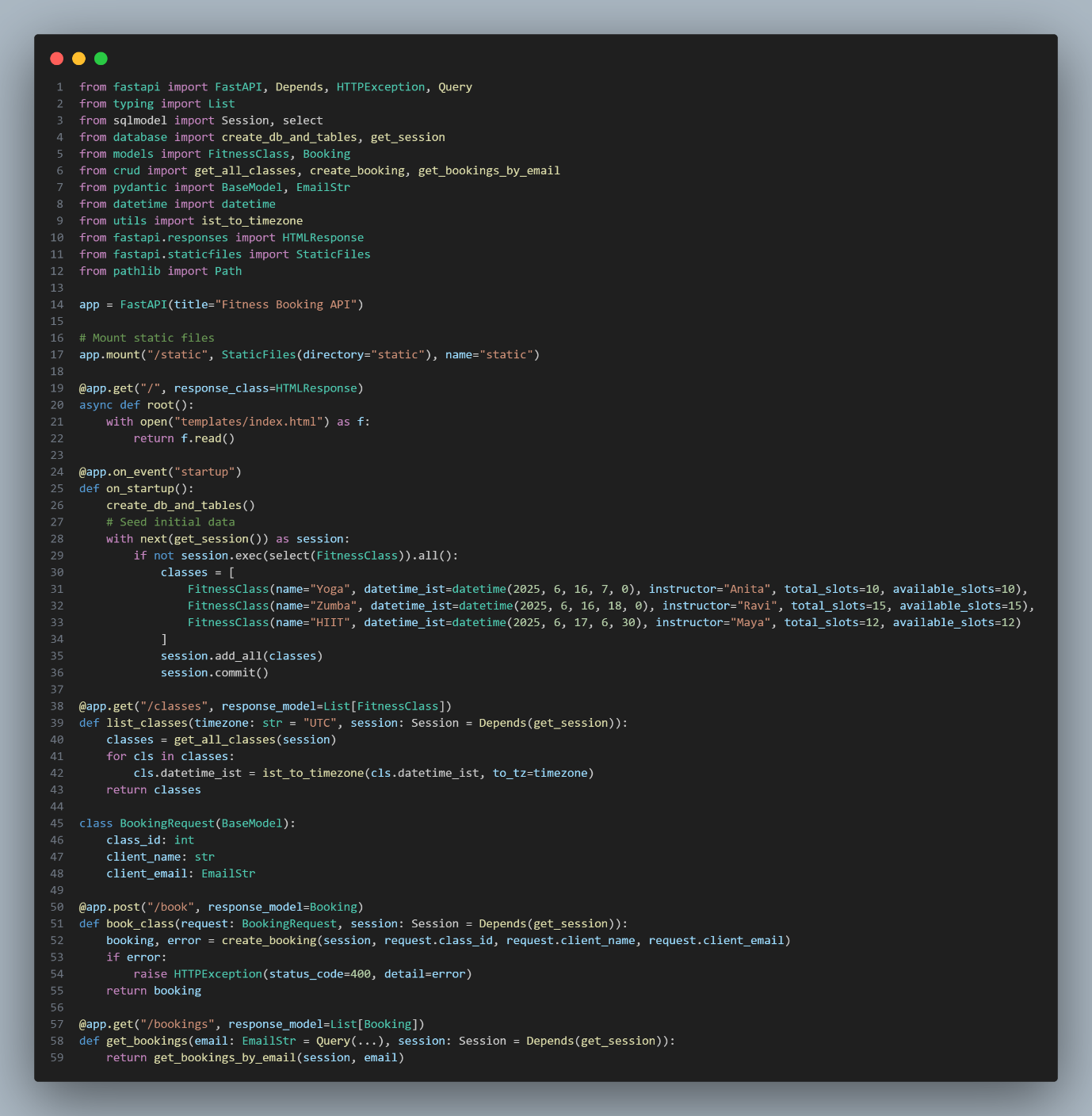


Configure database related thing in .env files.

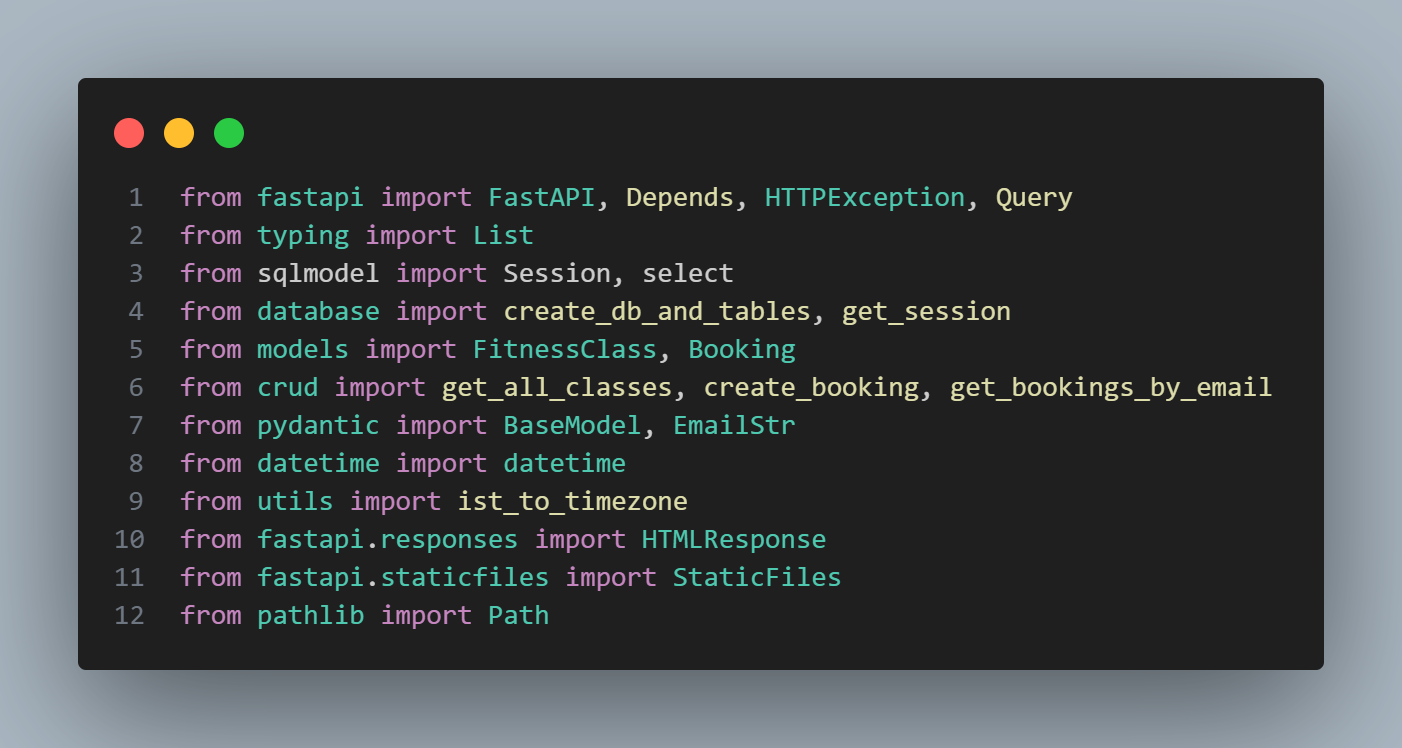


Database\_Url is present in Database.py file.

**Explanation of Main.py file code**



**Importing Required Modules**



These imports include:

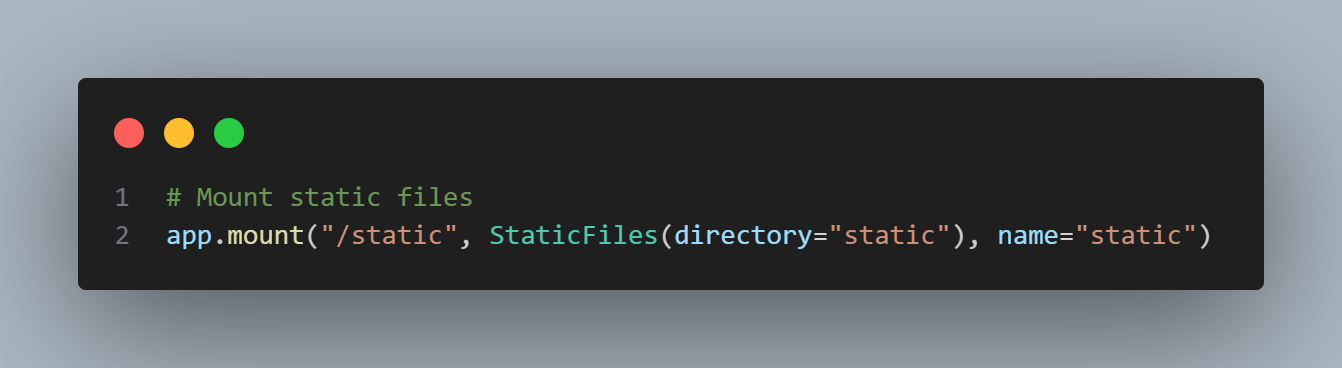
* FastAPI utilities (routing, error handling, dependencies).
* SQLModel functions for database sessions.
* Custom files: database.py, models.py, crud.py, utils.py.
* Pydantic classes for input validation.
* Standard libraries for date and file handling.

**Create FastAPI App Instance**



* Initializes a FastAPI application with the title "Fitness Booking API".

**Mount Static Files**



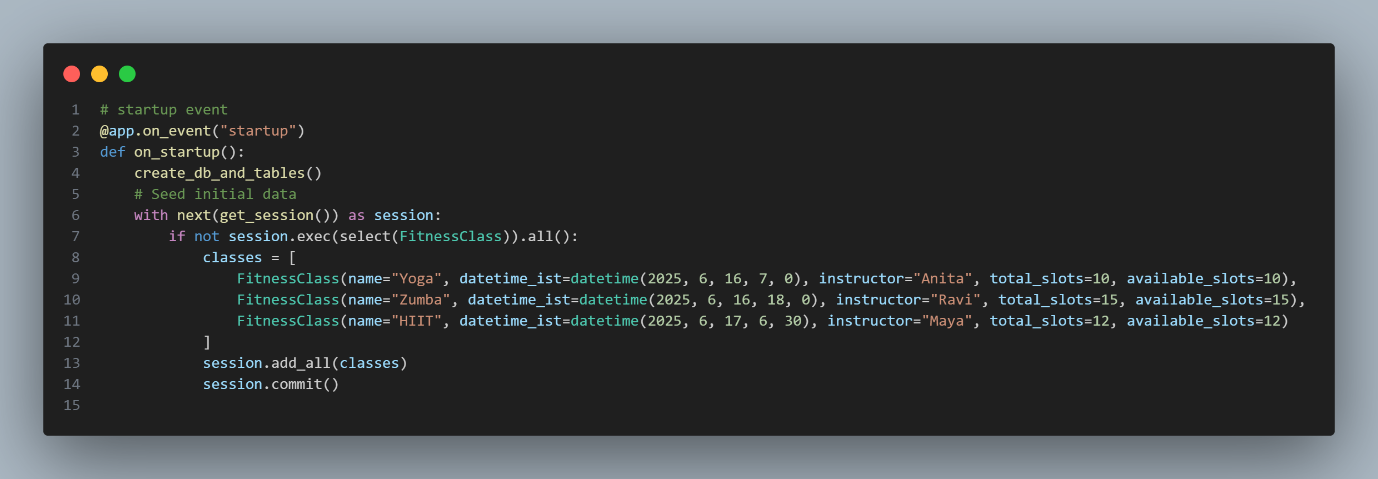
* Serves files in the static/ directory when the URL starts with /static.

**Root Route (Homepage)**



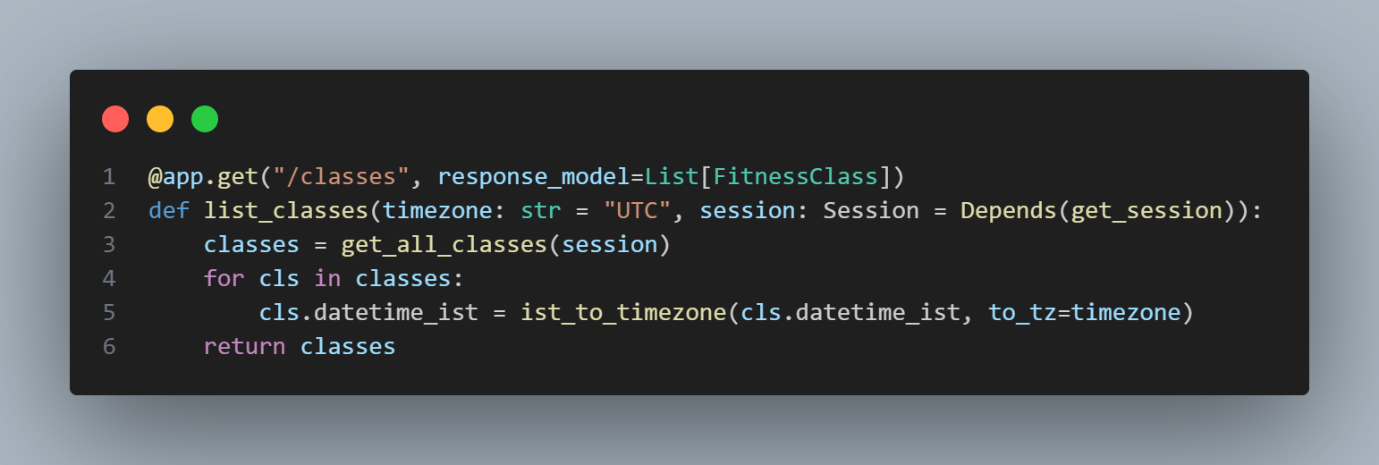
* Serves the index.html file from the templates/ folder when the root URL (/) is accessed.

**Startup Event**



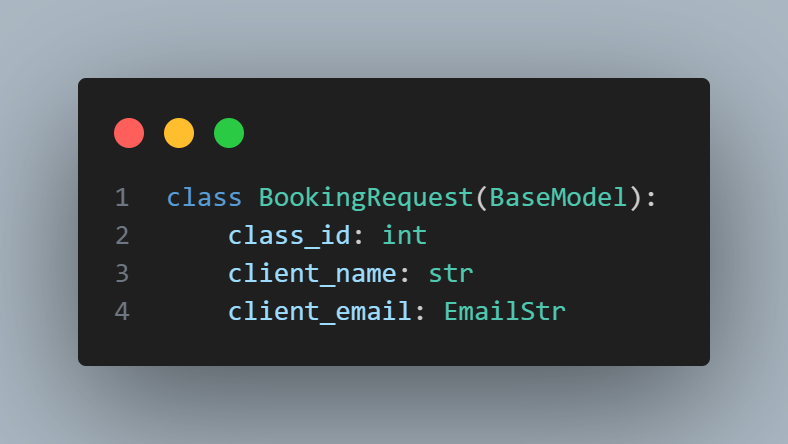
* Runs when the app starts.
* Creates database tables (if not already present).
* Seeds three sample fitness classes into the database if it's empty.

**Get All Classes**



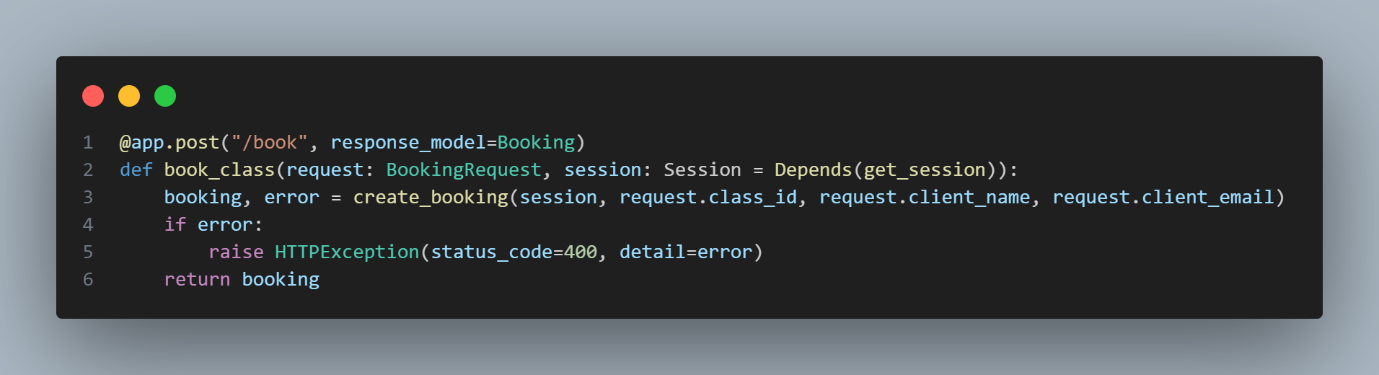
* Endpoint: /classes
* Fetches all classes from the database.
* Converts each class time from IST to the requested timezone.
* Returns a list of class details.

**Booking Request Model**



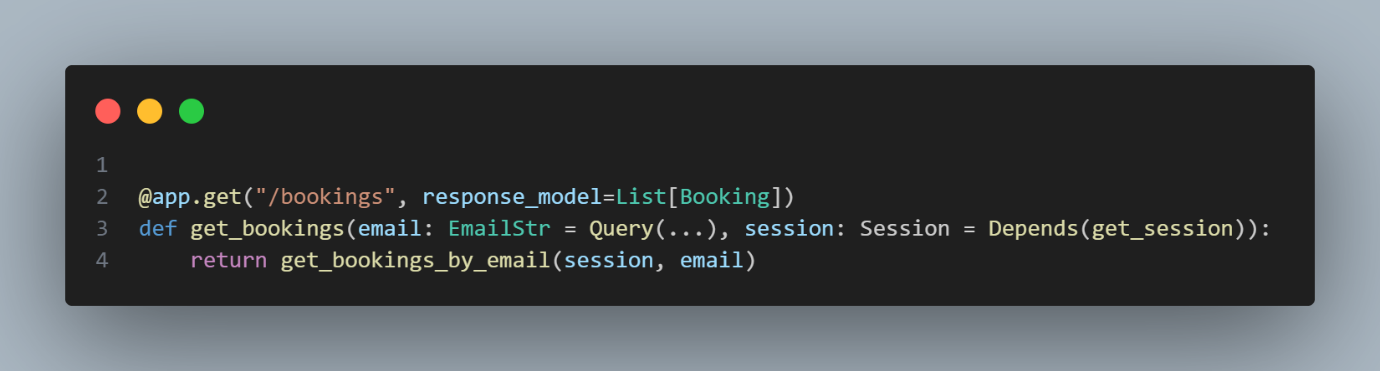
* Defines the format of the data required to book a class.
* Includes class ID, client name, and client email.

**Book a Class**



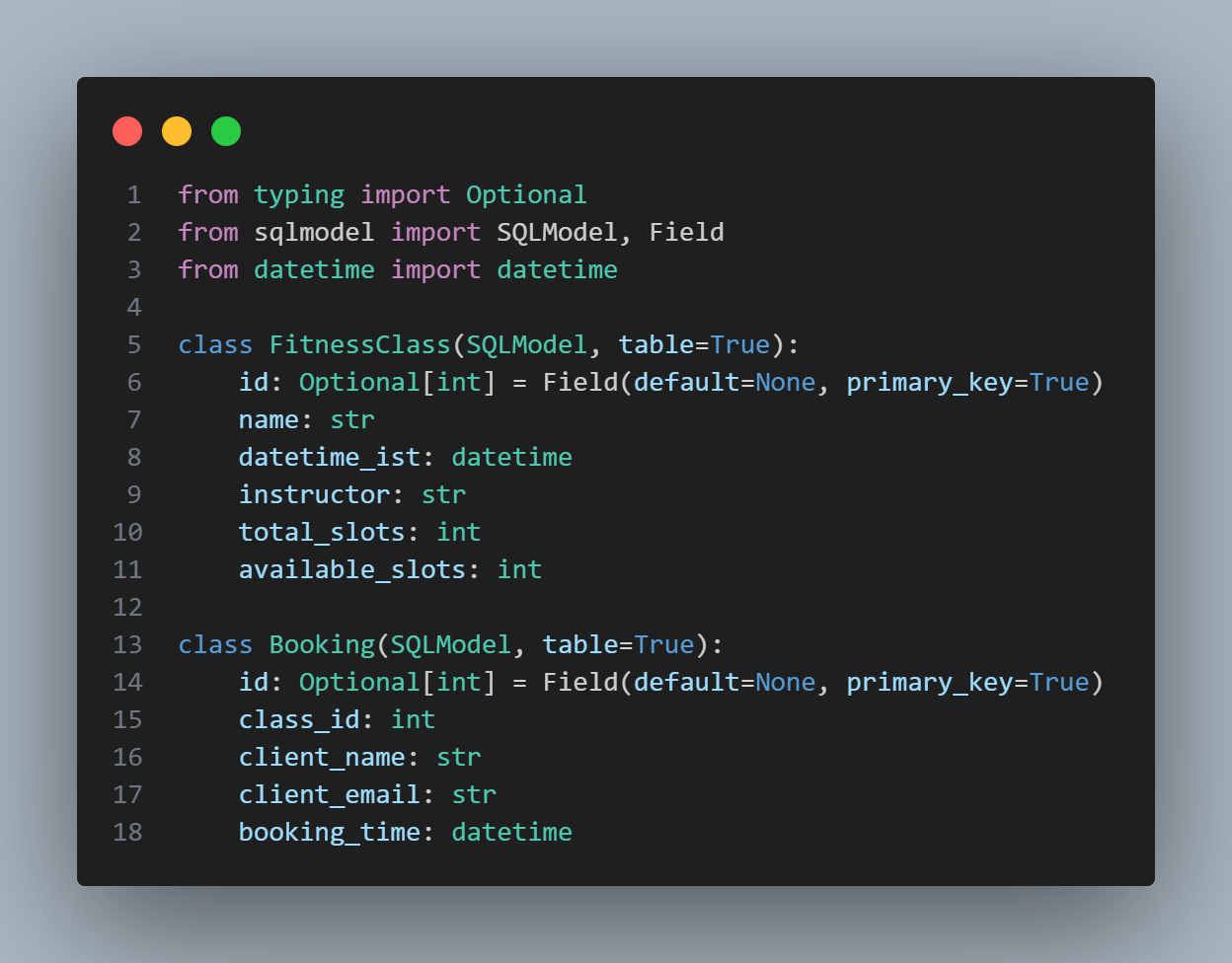
* Endpoint: /book
* Accepts booking request using the BookingRequest model.
* Calls create\_booking() to process the request.
* Returns the booking details if successful, else raises an error.

**Get All Bookings by Email**



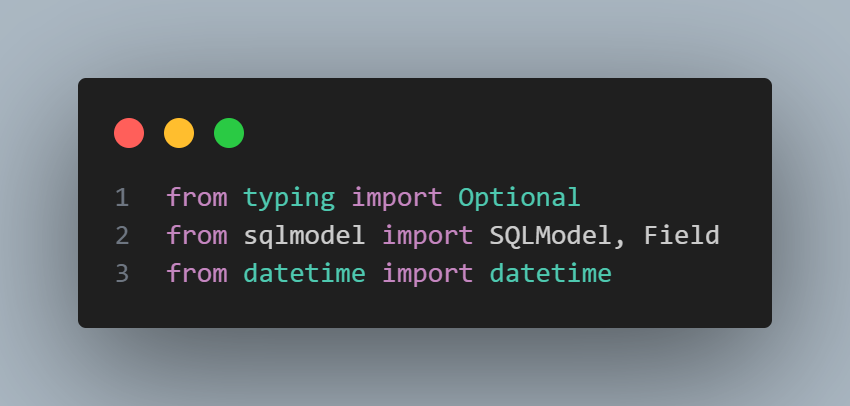
* Endpoint: /bookings
* Requires an email query parameter.
* Returns all bookings associated with that email.

**Models.py File Explanation**

****

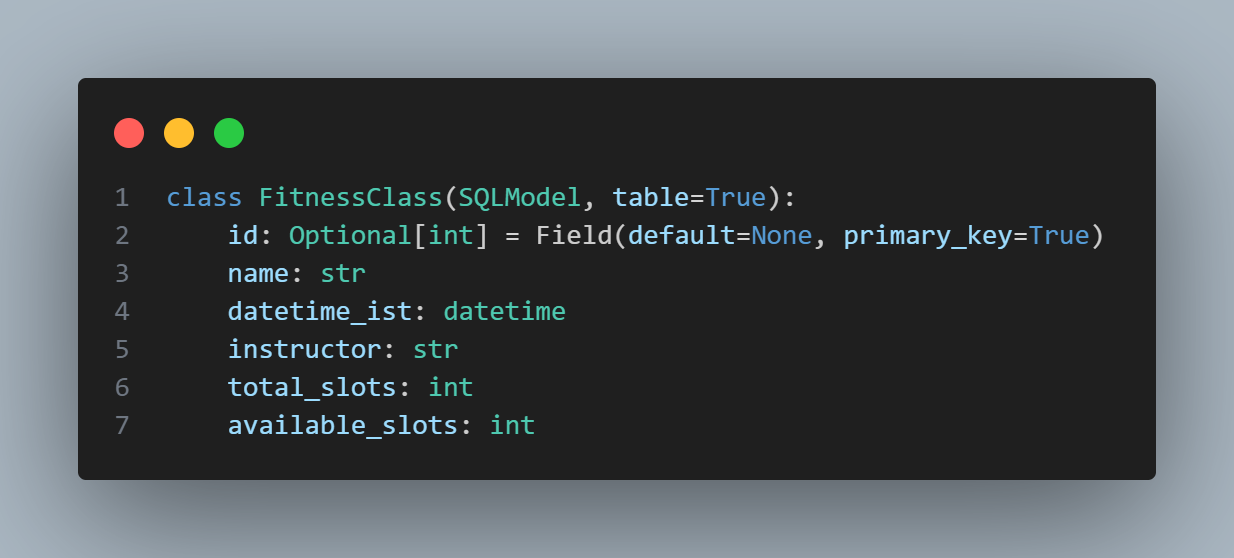
This file defines the database models using **SQLModel**, which combines features of **SQLAlchemy** and **Pydantic** for easy ORM and validation

**Imports**

****

* Optional: Allows nullable fields like id before database insertion.
* SQLModel and Field: Used to define table schemas.
* datetime: For handling date and time fields like class time or booking time.

**FitnessClass Model**

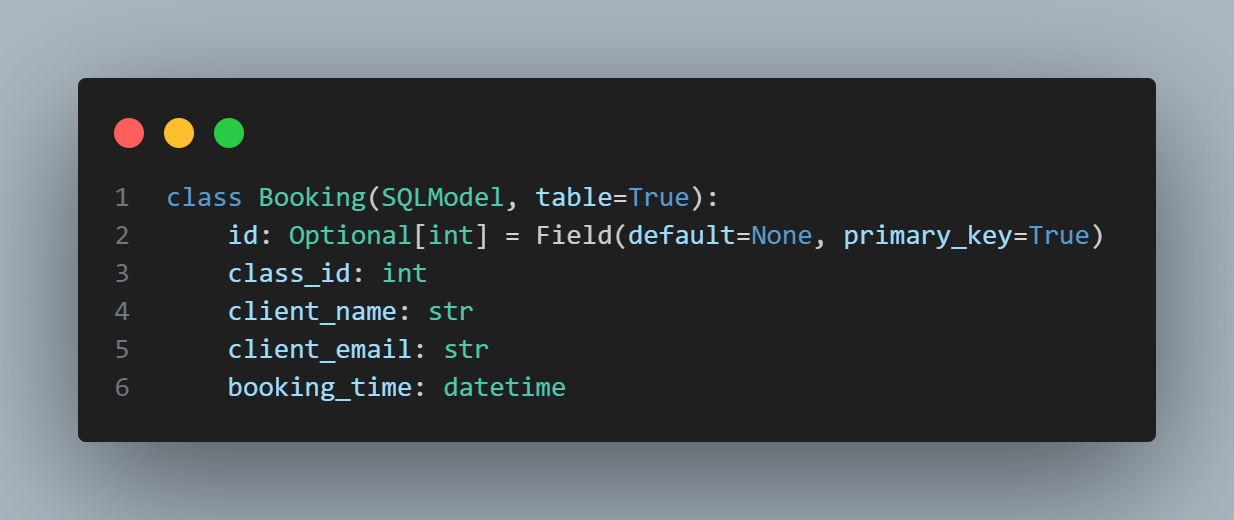
****

**Purpose:** Represents a fitness class (Yoga, Zumba, HIIT, etc.)

**Fields:**

* id: Primary key (auto-incremented).
* name: Name of the class.
* datetime\_ist: Date and time of the class (stored in IST).
* instructor: Name of the instructor.
* total\_slots: Total number of spots available.
* available\_slots: Current number of unbooked spots.

**Booking Model**

****

**Purpose:** Stores individual bookings made by clients.

**Fields:**

* id: Primary key.
* class\_id: Foreign key to FitnessClass.id (links the booking to a class).
* client\_name: Name of the person booking.
* client\_email: Email of the person booking.
* booking\_time: Timestamp when the booking was made.

**Database.py file explanation**

****

**Imports**

* SQLModel: For defining models and generating database schema.
* create\_engine, Session: For setting up the database connection and sessions.
* os: To access environment variables.
* load\_dotenv: Loads variables from a .env file into the environment.

**Load Environment Variables**

* Loads database configuration values from a .env file.
* Ensures sensitive data like username and password are **not hardcoded**.

**Create Database URL**

* Constructs a PostgreSQL connection URL dynamically using the loaded variables.

Create Tables Function

* Automatically creates all tables defined in models.py if they don't already exist.

**Session Generator**

* Provides a session object for interacting with the database.
* Used in API routes with FastAPI's Depends() for automatic session handling.

**crud.py file explanation**

****

**Imports**

* Session: Used for querying and modifying the database.
* select: Used to create SQL-style queries.
* FitnessClass, Booking: The two models defined in models.py.
* datetime: For adding a timestamp to new bookings.

**Get All Classes**

* Returns a list of all fitness classes from the database.
* Used in the /classes API endpoint.

**Get Class by ID**

* Returns a single FitnessClass object by its ID.
* Used before creating a booking to check if the class exists.

**Create Booking**

* **Step 1:** Checks if the class exists.
* **Step 2:** Checks if there are available slots.
* **Step 3:** Reduces available slots by 1.
* **Step 4:** Creates a new booking with name, email, and timestamp.
* **Step 5:** Saves the updated class and new booking to the database.

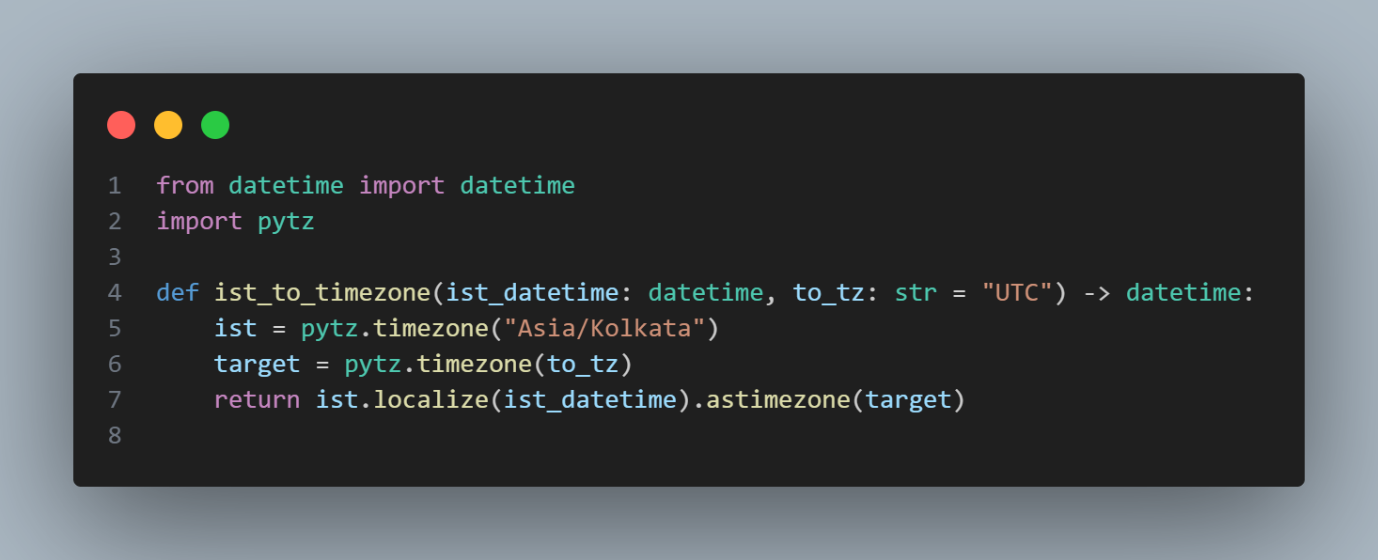
**Returns:**

* booking, None if successful.
* None, "error message" if class not found or overbooked.

**Get Bookings by Email**

* Returns a list of all bookings made by the given email address.
* Used in the /bookings API endpoint.

**utils.py file explanation**

****

This file contains **utility functions** that support your main application logic. In this case, it handles **timezone conversion**.

**Imports**

* datetime: Used for working with date and time values.
* pytz: A Python library for accurate timezone conversions.

**Timezone Conversion Function**

**Purpose:**

* Converts a datetime from **IST (Indian Standard Time)** to any other timezone (default is **UTC**).

**Parameters:**

* ist\_datetime: A datetime object assumed to be in IST.
* to\_tz: A string representing the target timezone (e.g., "Europe/London", "America/New\_York").

**Steps:**

1. Define the IST and target timezones using pytz.
2. Localize the input datetime to IST (attach timezone info).
3. Convert that IST time to the desired target timezone.

**Example:**

If a class is scheduled for **7:00 AM IST**, converting it to UTC would return **1:30 AM UTC**.

**Requirement Files**



**Docker File Explanation**

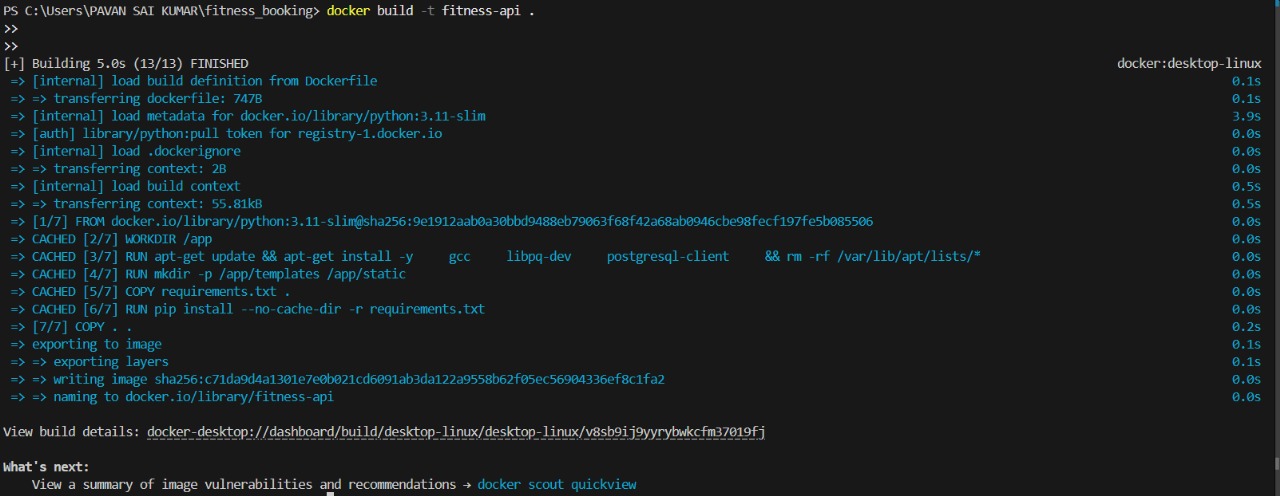


* **Base Image**: Uses a slim version of Python 3.11 to keep the image lightweight and efficient.
* **Working Directory**: Sets the working directory inside the container to /app, where the app files will reside.
* **Install Dependencies**: Installs system-level tools like GCC (for compiling), PostgreSQL client, and required development libraries for connecting to a PostgreSQL database.
* **Create Directories**: Prepares necessary folders like /templates and /static for the frontend files used by FastAPI.
* **Copy Dependency File**: Copies only the requirements.txt file first to take advantage of Docker's caching system.
* **Install Python Packages**: Installs all Python dependencies listed in requirements.txt using pip.
* **Copy Application Code**: Copies the remaining project files into the container to complete the setup.
* **Expose Port**: Opens port 8000 in the container so the FastAPI application can be accessed from outside.
* **Start Application**: Runs the FastAPI app using Uvicorn as the ASGI server, listening on all interfaces on port 8000.

**1. Build a Docker Image**

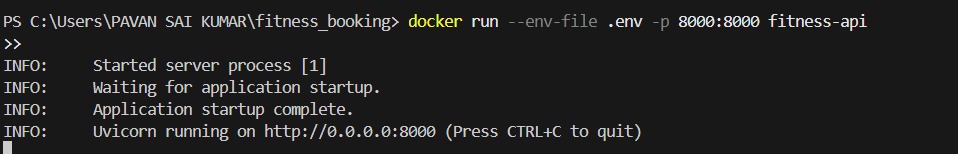
To build a Docker image from a Dockerfile:

* Command == docker build -t <image-name>:<tag> .



2. **Run the Docker Image (Create & Start Container)**

* Command == docker run -d -p <host-port>:<container-port> --name <container-name> <image-name>:<tag>



**3. List All Docker Images**

* Command == docker images

**4. List Running Containers**

* Command ==docker ps

To list **all containers** (including stopped ones):

* docker ps -a

**Pushing Docker Image to Docker Hub — Step-by-Step Guide**

* Prerequisites
* Docker must be installed and running on your machine.
* You must have a Docker Hub account.
* You should have a Docker image built locally.

Step 1: Login to Docker Hub

Use the following command to log in via CLI:

Command == docker login

* Enter your Docker Hub username and password when prompted.

If you’re using Docker Desktop, you can also log in via the GUI.

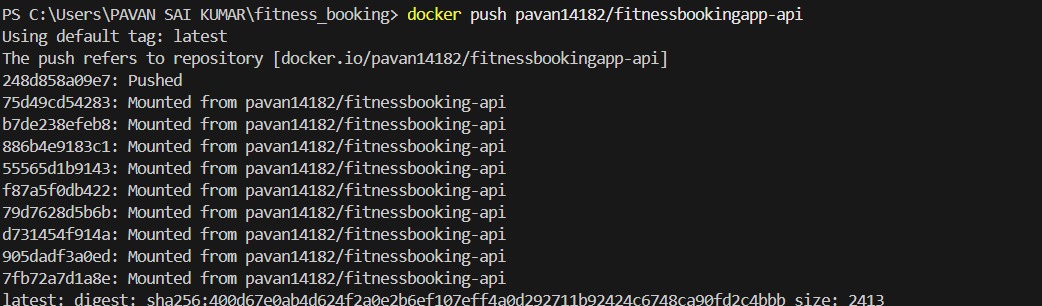
Step 2: Tag the Docker Image

You must tag your image with your Docker Hub username and repository name before pushing.

Command == docker tag <local-image-name>:<tag> <dockerhub-username>/<repository-name>:<tag>

Step 3: Push the Image to Docker Hub

Now push the tagged image:

Command == docker push <dockerhub-username>/<repository-name>:<tag>  
  


⏳ Wait for the upload to complete. Once done, the image will be visible on your Docker Hub account under Repositories.

Step 4: Verify the Image on Docker Hub

1. Go to https://hub.docker.com
2. Login and navigate to Repositories
3. You’ll see your image listed under your username

Optional: Pull the Image from Docker Hub on Another System

You can pull the pushed image from anywhere using:

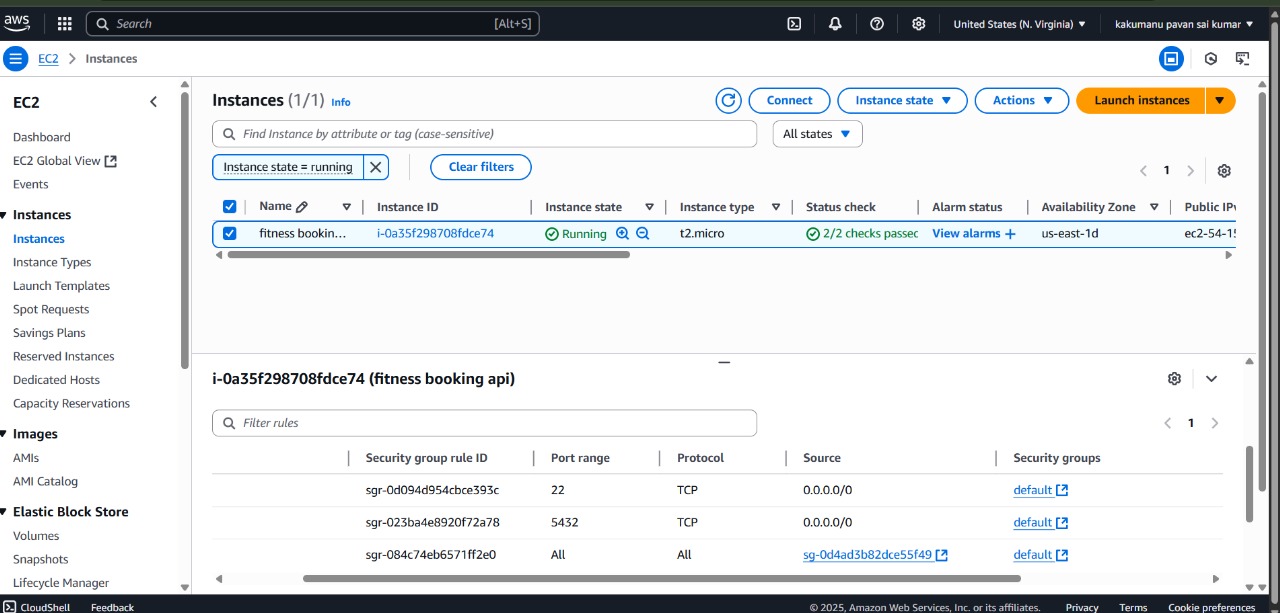
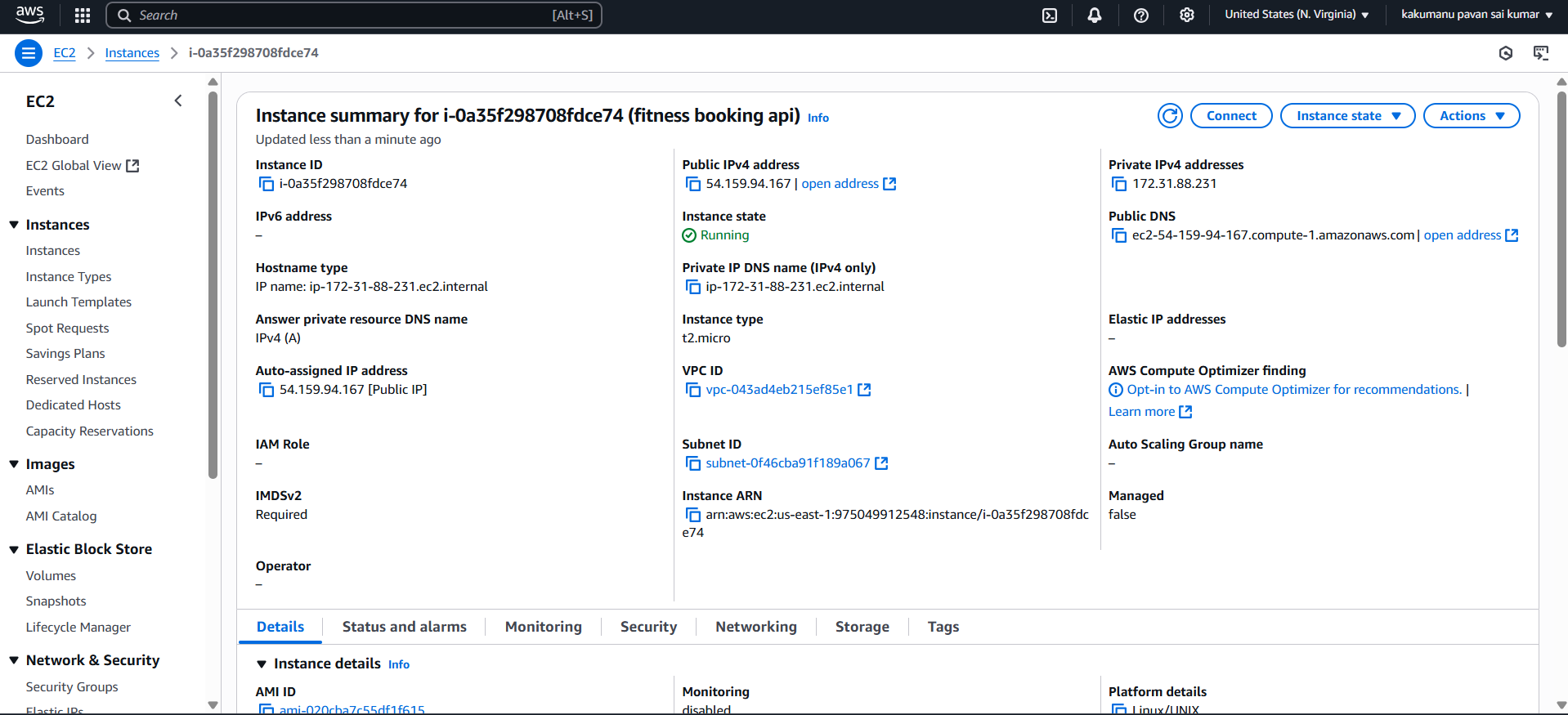
Command == docker pull <dockerhub-username>/<repository-name>:<tag>

**Deploy Fitness Booking API on EC2 Ubuntu with Docker**

* Prerequisites
* AWS account with EC2 access
* Security Group with port 80 open (HTTP access)
* Docker image: pavan14182/fitness-api

Step 1: Launch EC2 Ubuntu Instance

1. Go to [AWS EC2 Console](https://console.aws.amazon.com/ec2).
2. Click Launch Instance.
3. Set the following:
   * Name: fitness-booking-api
   * AMI: Ubuntu Server 22.04 LTS
   * Instance Type: t2.micro (Free tier eligible)
   * Key Pair: Create or use existing (needed for SSH)
   * Security Group: Add rule to allow HTTP (port 80) and SSH (port 22)
4. Click Launch Instance

 ****

Step 2: Connect to EC2 via SSH

Use the public IP or DNS provided in the EC2 dashboard.

Command == ssh -i "your-key.pem" ubuntu@<your-ec2-public-ip>

Step 3: Install Docker on Ubuntu EC2

https://docs.docker.com/engine/install/ubuntu/  
by using this doc install the docker in the ubuntu ec2   
  
Step 4: Pull Docker Image

command == sudo docker pull pavan14182/fitness-api

Step 5: Create .env File

Inside your EC2 home directory:

Command == nano .env

Paste the following content:

DB\_HOST=fitness-booking.csrw8i0isikt.us-east-1.rds.amazonaws.com

DB\_PORT=5432

DB\_NAME=fitnessdb

DB\_USER=postgres

DB\_PASSWORD=pavan1234

Save and exit (Ctrl+O, Enter, Ctrl+X)

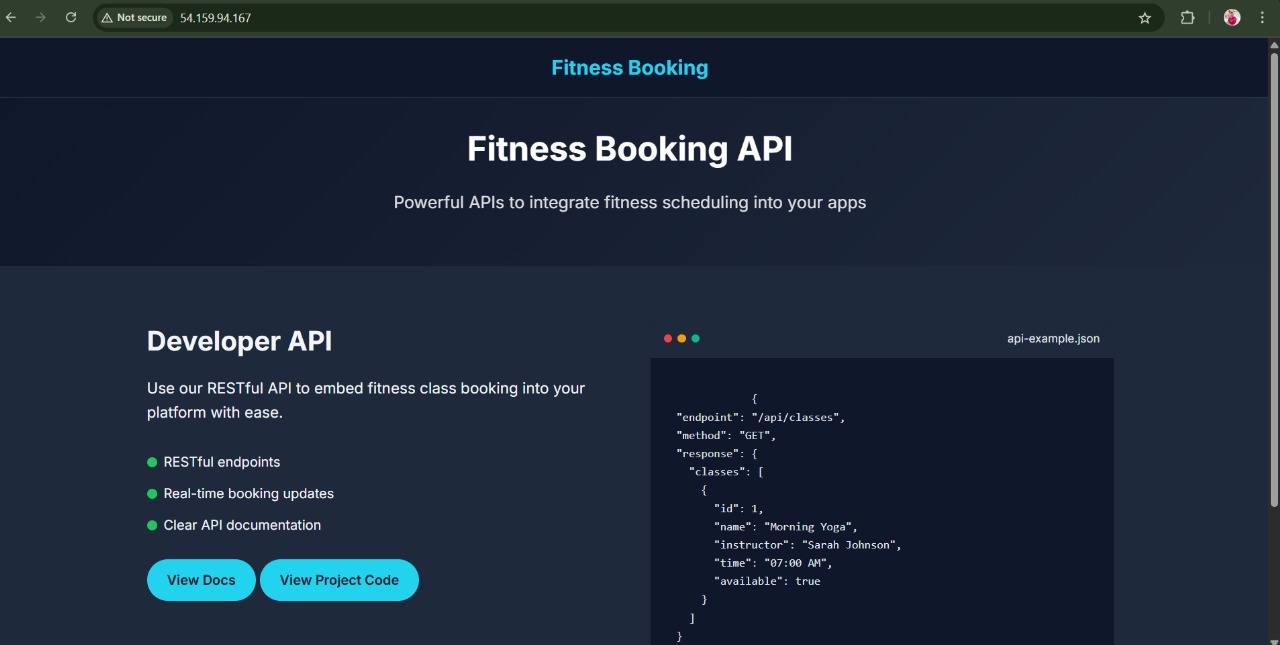
Step 6: Run Docker Container

Command sudo docker run --env-file .env -d -p 80:8000 pavan14182/fitness-api

Step 7: Access the Application

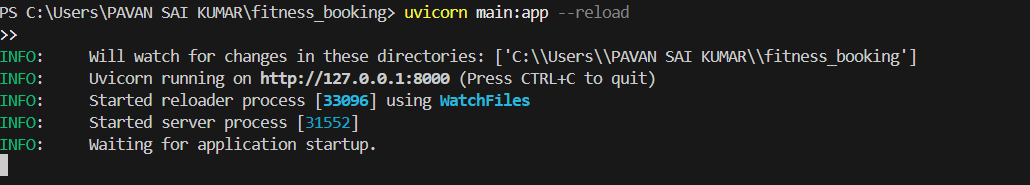
Open your browser and go to:  
http://<your-ec2-public-ip>

You should see your Fitness Booking API running!



**Fitness Booking API – FastAPI Project Walkthrough**

Running FastAPI Server (uvicorn main:app --reload)

****

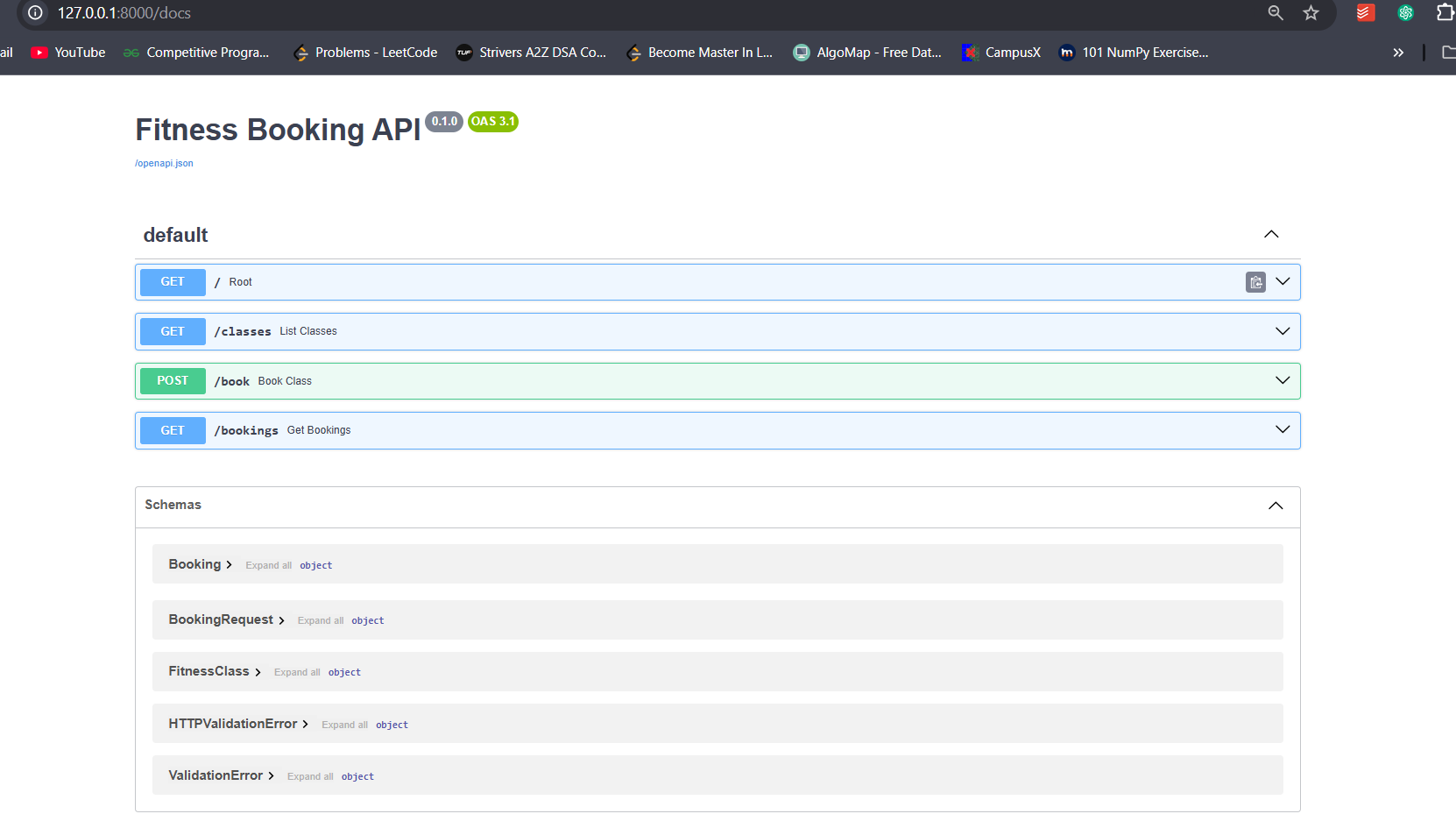
* This shows the successful startup of the FastAPI server using Uvicorn.
* It's running at: http://127.0.0.1:8000.
* The --reload flag means it will auto-reload if code changes, useful during development.

**Home Page of Fitness Booking App**

* This is the custom HTML landing page you created for the project.
* Highlights:
  + Project name: Fitness Booking
  + Features: RESTful endpoints, real-time updates, and documentation.
  + Buttons like View Docs and View Project Code (optional links).



**Swagger UI (/docs endpoint)**



* Auto-generated docs provided by FastAPI using **OpenAPI 3.1**.
* Shows 3 main endpoints:
  1. GET /classes – View all fitness classes.
  2. POST /book – Book a fitness class.
  3. GET /bookings?email=... – See all bookings for a user.
* Each endpoint is testable right from the browser.
* Schema section explains request/response models like Booking, BookingRequest, etc

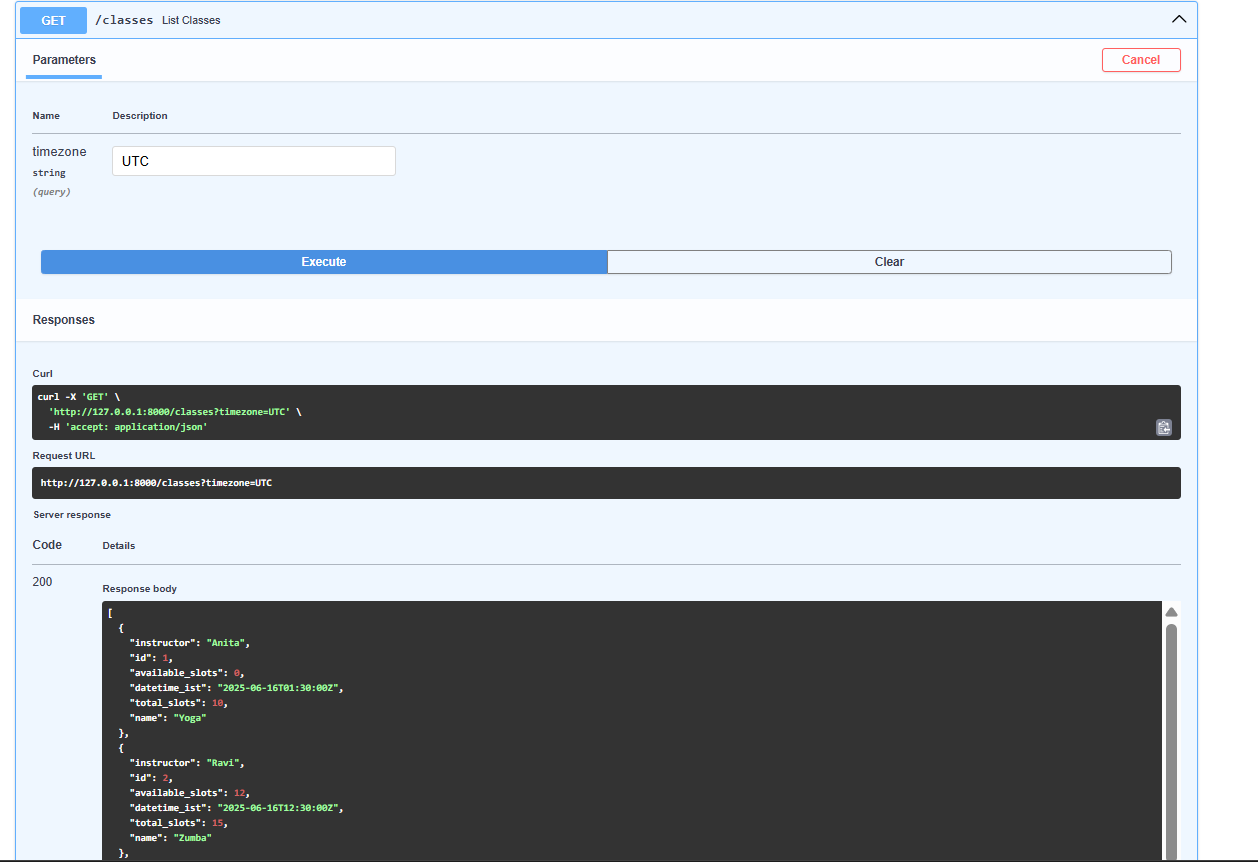
**Testing /classes Endpoint with Different Timezones**

**Objective:**  
To verify if the /classes API correctly converts and displays datetime information based on the provided timezone (e.g., UTC, Asia/Kolkata).

**Key Results:**

* The API returns datetime in UTC format when timezone=UTC is used.
* When timezone=Asia/Kolkata is used, the response datetime is adjusted by +05:30 hours, confirming proper timezone conversion.

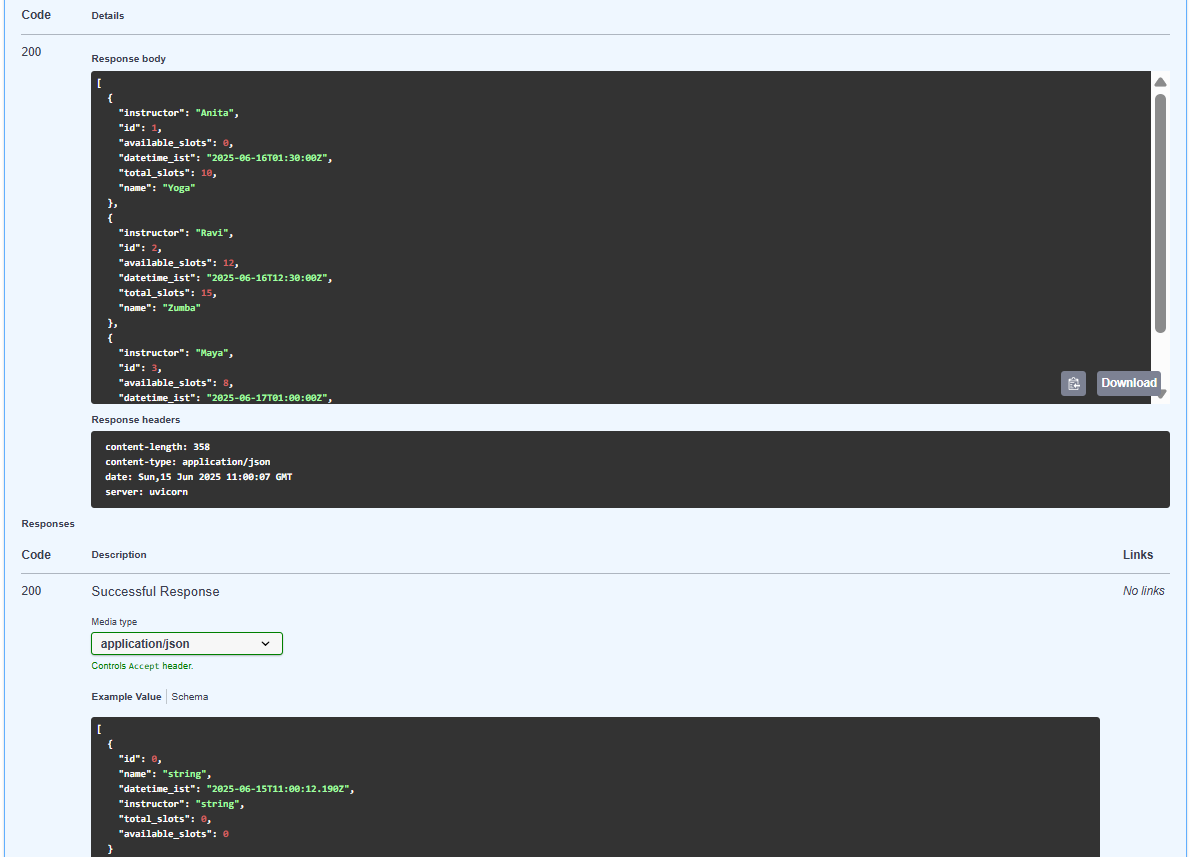
**Purpose**: Testing the /classes endpoint with timezone=UTC



**Details**:

* Shows a list of classes returned by the API with datetime in **UTC** (e.g., "2025-06-16T11:30:00Z").
* Each class has details like:
  + id, name, instructor, available\_slots, and total\_slots

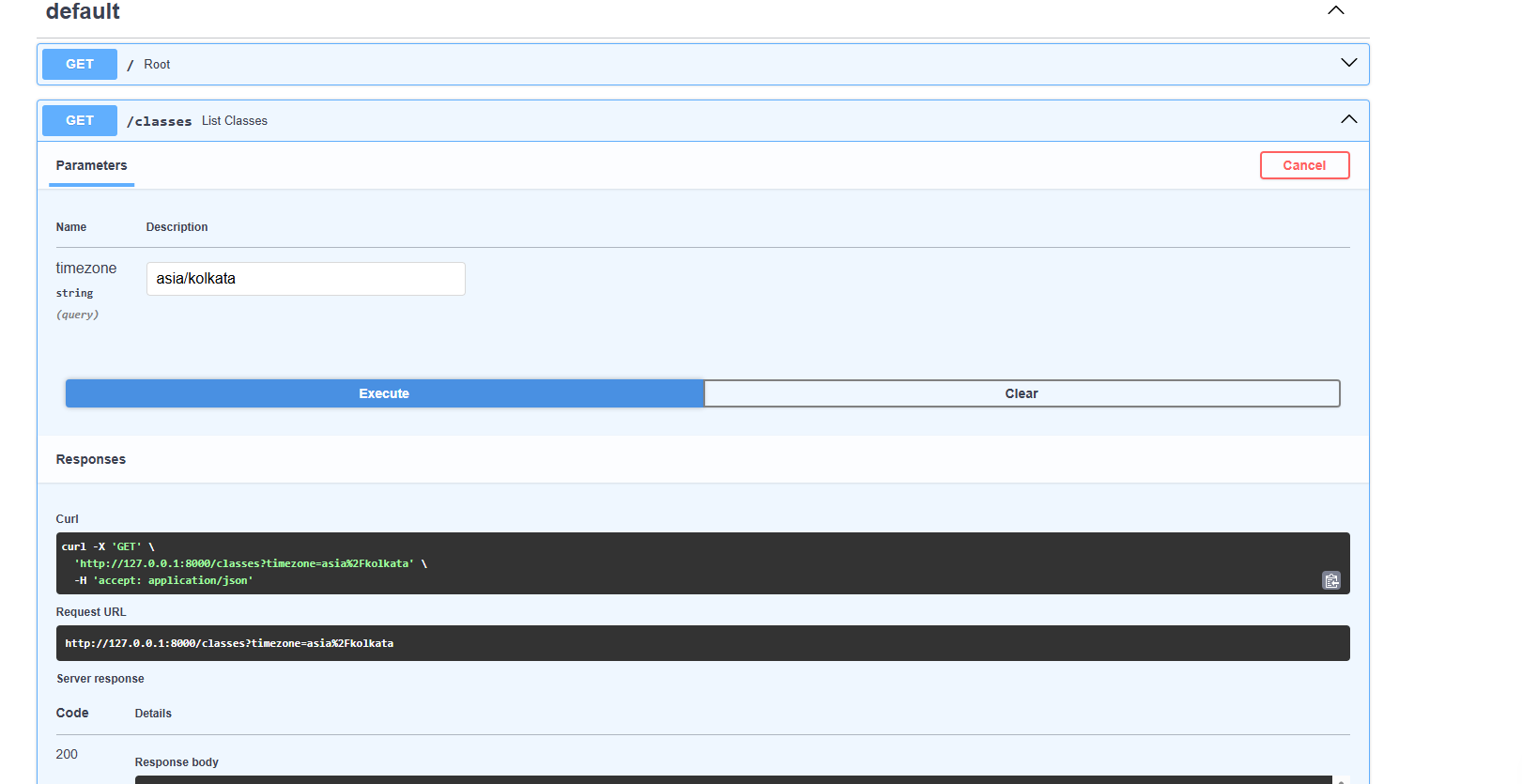
**Response**



Purpose: Continuation of the previous UTC response (scroll down)  
Details:

* More class entries shown.
* Final response confirms data format (JSON) and proper UTC datetimes.
* Headers show content type is application/json and server is uvicorn.

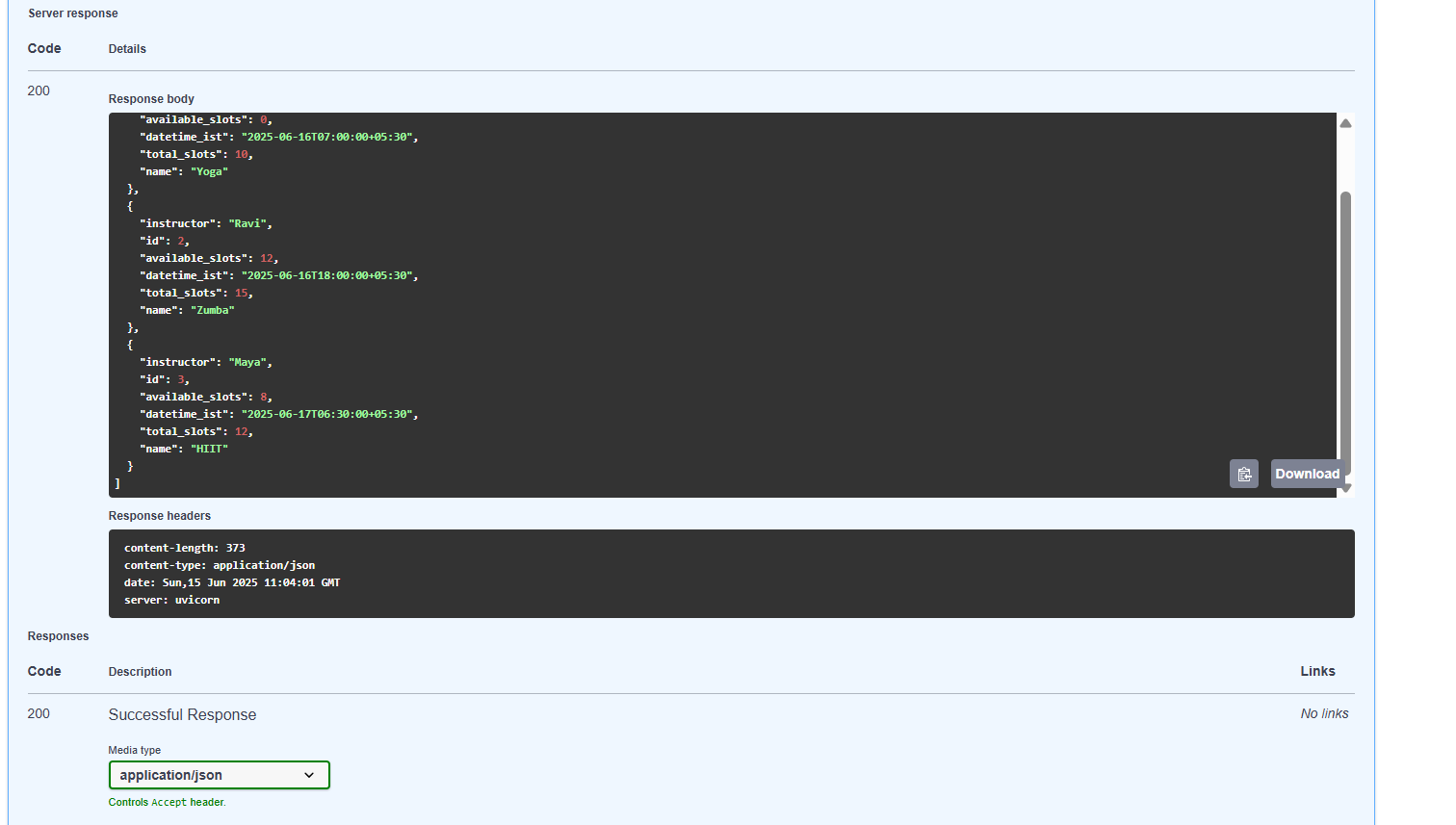
**Purpose**: Testing the /classes endpoint with timezone=Asia/Kolkata



**Details:**

* Shows that the datetime\_ist values are now converted to **IST** (UTC+5:30).
  + Example: "2025-06-16T17:00:00+05:30"
* Confirms that the time conversion is working based on the query parameter.

**response**



P**urpose**: Likely a summary or notes of the testing steps

**Details**:

* Describes the purpose of using different timezones (UTC, Asia/Kolkata) to test datetime formatting.
* Notes that UTC was converted by adding +5:30 to show IST correctly.

You can test with this time zones

Europe

* Europe/London
* Europe/Berlin
* Europe/Paris
* Europe/Moscow

America

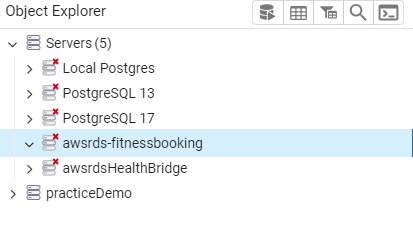
* America/Chicago
* America/New\_York
* America/Los\_Angeles
* America/Sao\_Paulo

Etc

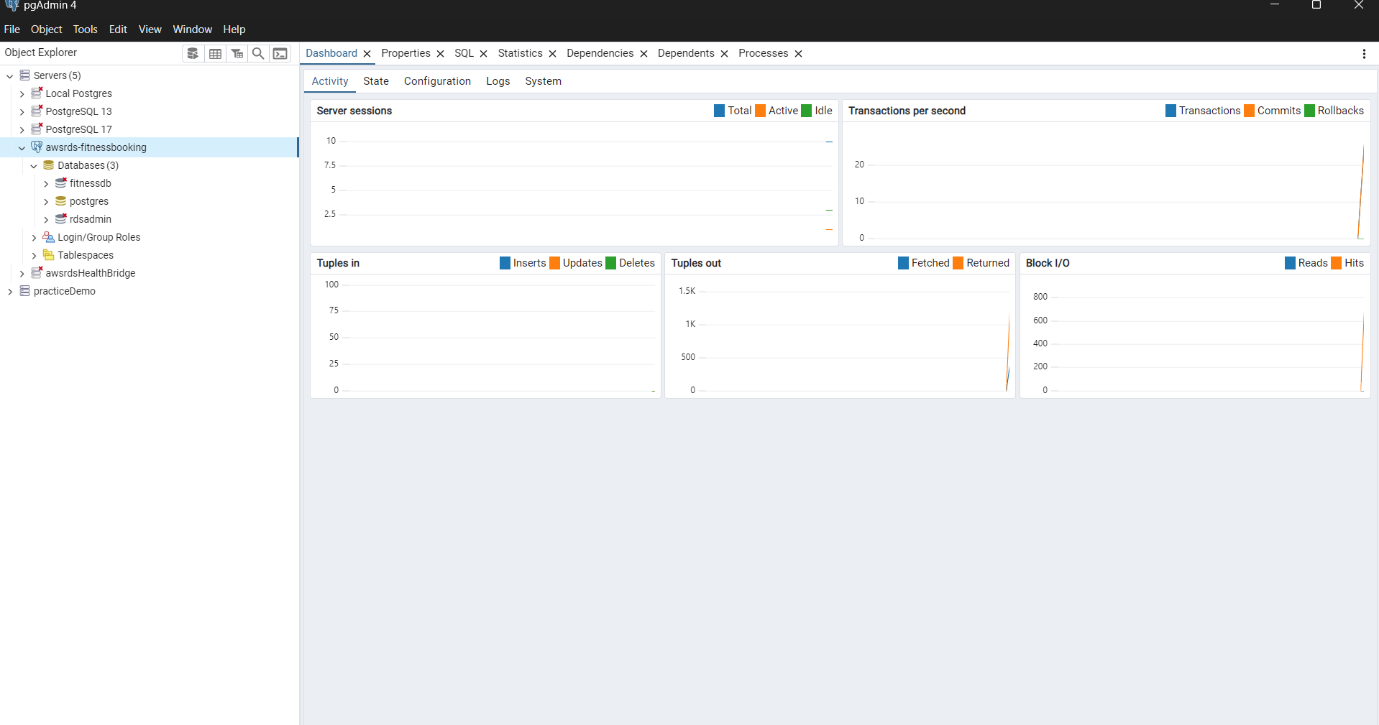
**Booking Api testing**

**PgAdmin(postgres) database connection Setup**

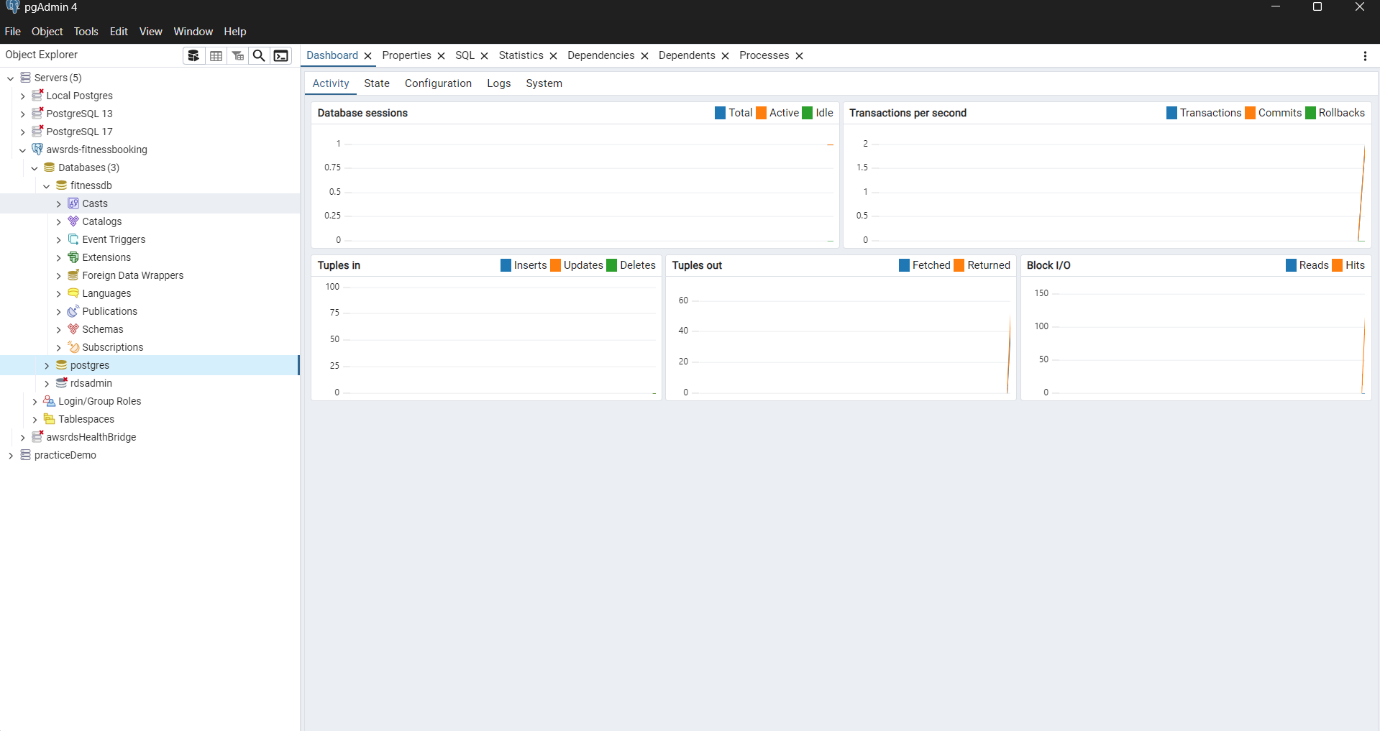
To connect to the PostgreSQL database using pgAdmin, open pgAdmin and look at the top left side — you will see the list of servers as shown in the image below.



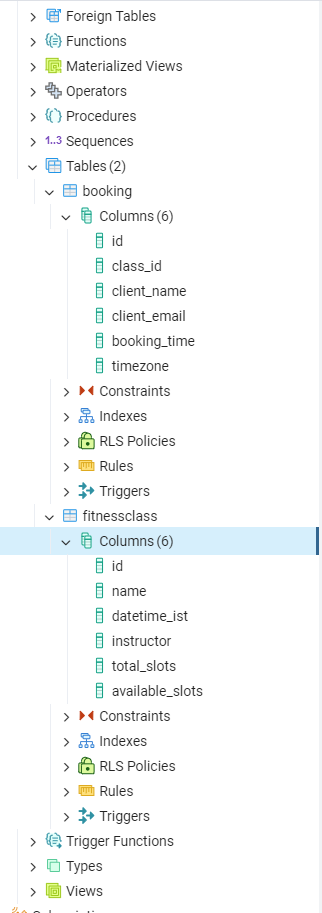
Next, create your own server just like I did in the image above. Then, enter your AWS credentials, including the endpoint, username, password, and other required details. Once it's done, you will be able to see the connection established, as shown below.



Inside the server, create a new database and name it as you wish. In the picture below, you can see that I named my database 'fitness\_database'.



Next, you will see the booking and fitnessclass tables inside your database. If they don’t appear, make sure to run your FastAPI application — it will automatically create and seed the tables

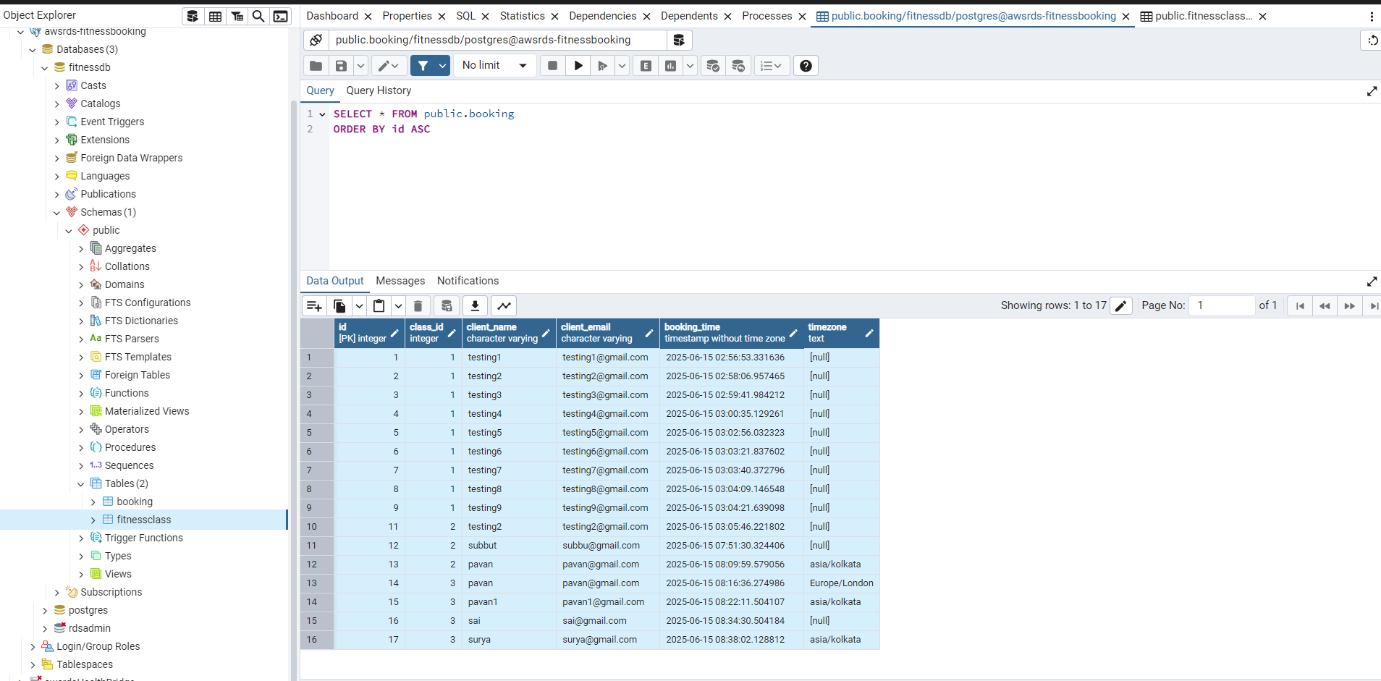


**Booking Table Data**

The booking table stores information about class reservations made by clients. Each row represents a booking record.

**Columns:**

* **id** – Unique booking ID (primary key)
* **class\_id** – Foreign key referencing the fitness class
* **client\_name** – Name of the client who booked
* **client\_email** – Email address of the client
* **booking\_time** – Timestamp of when the booking was made (in UTC)



**FitnessClass Table Data**

The fitnessclass table stores all available fitness class details. Each row represents a scheduled class offered by the studio.

**Columns:**

* **id** – Unique class ID (primary key)
* **name** – Name of the class (e.g., Yoga, Zumba, HIIT)
* **datetime\_ist** – Date and time of the class in IST (Indian Standard Time)
* **instructor** – Name of the instructor conducting the class
* **total\_slots** – Total number of available slots for the class
* **available\_slots** – Number of slots still available for booking

