

Build a small **backend API** for a **Service Membership System** (like a gym / coaching centre / salon).

We want to see:

- How you design **database tables**
- How you structure a **FastAPI** project
- How you handle **business logic**
- How you implement a simple **database trigger**

No frontend is required.

## 1. Tech Requirements

- **Language:** Python (3.9+)
- **Framework:** FastAPI
- **Database (SQL):**
  - Preferred: **PostgreSQL**
  - Acceptable: SQLite (only if Postgres is not easily available)
- **ORM:** Any (SQLAlchemy / SQLModel / Tortoise etc.)

## 2. Data Model (Minimum Tables)

Design at least these 4 tables:

### 2.1 Member

- `id` (primary key)
- `name`
- `phone`
- `join_date`
- `status` (e.g. `active`, `inactive`)
- `total_check_ins` (integer, **default 0**) (used by trigger)

### 2.2 Plan

- `id` (primary key)
- `name` (e.g. "Monthly", "Quarterly")
- `price`
- `duration_days` (integer)

### 2.3 Subscription

- `id` (primary key)
- `member_id` (FK → Member)
- `plan_id` (FK → Plan)
- `start_date`

- `end_date`

## 2.4 Attendance

- `id` (primary key)
- `member_id` (FK → Member)
- `check_in_time` (datetime)

You may add extra fields if you feel they are useful.

## 3. Required API Endpoints

You don't have to build a huge system; just make the core endpoints work cleanly.

### 3.1 Members

#### 1. POST /members

- Create a new member.
- Validate:
  - `name` is required
  - `phone` is required
- If you want, you can enforce `phone` unique.

#### 2. GET /members

- List all members.
- Optional query parameter: `status` (e.g. `/members?status=active`).

### 3.2 Plans

#### 3. POST /plans

- Create a new plan with `name`, `price`, `duration_days`.

#### 4. GET /plans

- List all plans.

### 3.3 Subscriptions

#### 5. POST /subscriptions

- Create a subscription for a member.
- Request body should include:
  - `member_id`
  - `plan_id`
  - `start_date`
- Backend should automatically set:
  - `end_date = start_date + duration_days` of the plan.

#### 6. GET /members/{member\_id}/current-subscription

- Return the **current active subscription** for that member, based on today's date between `start_date` and `end_date`.
- If no active subscription exists, return a clear message.

### 3.4 Attendance

#### 7. POST /attendance/check-in

- Request body:
  - `member_id`
- Logic:
  - Check if the member has an **active subscription** (same rule as above).
  - If active:

- Insert an Attendance record with `check_in_time = now`.
- If not active:
  - Return an error message like: "No active subscription for this member".

8. `GET /members/{member_id}/attendance`

- Return a list of all attendance records for that member.

## 4. Database Trigger

We want to see **one database-level trigger**.

### 4.1 New Column (already included)

In the **Member** table, you have:

- `total_check_ins` (integer, default 0)

### 4.2 Trigger Requirement

Implement a **database trigger** that keeps `total_check_ins` in sync:

Whenever a new row is inserted into **Attendance**, automatically increment `total_check_ins` of that member.

Concretely:

- Table: `attendance`
- Event: `AFTER INSERT`
- Action:
  - Look at `NEW.member_id`
  - Run:

```
UPDATE member SET total_check_ins = total_check_ins + 1
WHERE id = NEW.member_id;
```

### 4.3 Expectations by DB

- If using **PostgreSQL** (preferred):
  - Implement as a real trigger via SQL:
    - `CREATE FUNCTION ...`

- `CREATE TRIGGER ...`
- Put this SQL in:
  - a file like `triggers.sql`, or
  - an Alembic migration (if you use Alembic).
- If using **SQLite**:
  - SQLite also supports triggers, so you can still write:
    - `CREATE TRIGGER ...`
  - If you absolutely cannot use triggers in your environment:
    - Clearly explain in the README how you **would** do it at DB level,
    - You may simulate it in application code on insert (this is acceptable but will score less than a real DB trigger).

## 4.4 What to Submit About the Trigger

Please:

1. Include the **trigger SQL** in the repository (e.g. `triggers.sql` or migration file).
2. In `README.md`, add a short note:
  - Where the trigger is defined (which file)
  - What it does
  - How to run/apply it (if it's not automatic)

## 5. Non-Functional Requirements

- Use **Pydantic models** for request/response schemas.
- Do basic **input validation** (e.g. required fields, correct types).
- Handle errors cleanly:
  - 404 for not-found records (member/plan)
  - 400/422 for invalid input

Keep a **reasonable project structure**, e.g.:

```
app/  
  main.py          # FastAPI app  
  models.py        # ORM models  
  schemas.py       # Pydantic models  
  database.py      # DB session / engine  
  routers/  
    members.py  
    plans.py  
    subscriptions.py  
    attendance.py
```

The exact structure is up to you. We just want it to be clean and readable.

## 6. What to Submit

Please send:

1. **GitHub / GitLab repository link** (preferred), or a zip with:
  - Source code
  - `requirements.txt` (or `pyproject.toml`)
2. A **README.md** that includes:
  - How to set up the project:
    - Create and activate virtualenv
    - Install dependencies
    - Configure **DATABASE\_URL** (Postgres or SQLite)
  - How to create tables & apply the trigger (migrations or running SQL scripts)
  - How to start the app:
    - e.g. `uvicorn app.main:app --reload`
  - Example requests for key endpoints  
(e.g. JSON body examples or simple curl/Postman screenshots).

## 7. Evaluation Criteria

We will evaluate on:

1. **Correctness & completeness** of required endpoints
2. **Data modeling & relationships** (Member, Plan, Subscription, Attendance)
3. **DB trigger implementation** (real DB trigger preferred)
4. **Code structure & readability**
5. **Error handling & validation**
6. **Documentation (README clarity)**

Bonus points for:

- Nice project structure
- Simple tests
- Small extra touches (search, filters, etc.)