COMP3005 - Final Project Report

(excluding ER Diagram and ER to relational mapping)

Abdul Malik
101269165

Submitted to:
Professor Abdelghny Orogat
COMP 3005 - Database Management Systems
School of Computer Science
Carleton University

December 1, 2025

INTRODUCTION/IMPORTANT INFO:

This report documents some important information for my implementation of the COMP3005 final project, a Health & Fitness Club Management System backed by PostgreSQL and a Python/SQLAlchemy ORM application layer.

I am implementing the project solo and here is what my system takes care of functionally:

Member role

- USER REGISTRATION
    - Register as a new user by providing their basic info (name, gender, email, phone), plus optional fitness details like date of birth, goal weight, and current weight. The registration function makes sure the gender matches the allowed values and that email/phone are unique before inserting anything.

- PROFILE MANAGEMENT
    - Update an existing member's profile (email and phone number) while checking for uniqueness

- DASHBOARD
    - View a personal "member dashboard" of upcoming sessions via a database VIEW

- PT SESSION SCHEDULING
    - Book personal training (PT) sessions while checking that the trainer is actually available in the specific time window, and member/trainer double booking

Trainer role

- SET AVAILABILITY
    - Lets trainer set time windows for their availabilities. The function rejects anything where the end time is before the start, where the start is in the past, or where the new slot overlaps an existing availability window for that trainer.

- SCHEDULE VIEW
    - View all the upcoming schedules & classes for a particular trainer

Admin role

- ROOM BOOKING
    - Create new rooms with a given name and max capacity. The function trims the room name, enforces a positive capacity, checks that the admin_id exists, and blocks duplicate room names so you don't end up with two "Weight Room" entries.

- CLASS MANAGEMENT
    - For CLASS sessions, I have an admin-side function that lets you create a class for a specific trainer, in a specific room, over a specific time window. In my implementation, the session type is always forced to "CLASS" so it's clear this path is only for group classes. I also validate the capacity: it has to be a positive number, and it can't be larger than the room's own max_capacity, so you can't overbook the space. Before inserting anything, the code checks that the trainer actually has an availability block that fully covers the requested start and end time, and that the trainer isn't already scheduled to teach something else that overlaps in that window (even in another room). On top of the Python checks, I also rely on a PostgreSQL trigger (prevent_room_overlap) to catch any overlapping sessions in the same room at the database level, so even if something slipped through in the application layer, the DB would still reject the conflict

ORM EXPLANATION:

In this project I use SQLAlchemy's ORM so that every table in the Health & Fitness Club schema is represented as a Python class instead of writing raw SQL everywhere. Each class maps directly to one of the main entities in my ER diagram – for example Member, Trainer, Admin_staff, Room, Session, and TrainerAvailability. The Columns are declared as normal Python attributes with types and constraints, and SQLAlchemy takes care of generating the correct DDL and queries behind the scenes. An example of the Member class for instance looks like this:

```python
class Member(Base):
    __tablename__ = "member"

    # primary key and attributes
    member_id = Column(Integer, primary_key=True, autoincrement=True)
    goal_weight = Column(Numeric(5, 2))
    current_weight = Column(Numeric(5,  2))
    gender = Column(String(20), nullable=False)
    first_name = Column(String(50), nullable=False)
    last_name = Column(String(50), nullable=False)
    year = Column(Integer)
    month = Column(Integer)
    day = Column(Integer)
    phone_number = Column(String(20), unique=True)
    email = Column(String(255), nullable=False, unique=True)
```

Relationships between entities are also handled at the ORM level. For example, a Session object links a trainer, a room, and (optionally) a single member for PT sessions. Instead of manually joining tables, the service layer can follow these relationships directly in Python, which keeps the higher-level logic (like booking PT, creating classes, or printing dashboards) much more clean:

```python
class Session(Base):
    __tablename__ = "session"

    # primary key and attributes
    session_id = Column(Integer, primary_key=True, autoincrement=True)
    session_type = Column(String(10), nullable=False)
    start_date_time = Column(DateTime, nullable=False)
    end_date_time = Column(DateTime, nullable=False)
    max_capacity = Column(Integer, nullable=False)

    # foreign keys
    room_id = Column(Integer, ForeignKey("room.room_id"), nullable=False)
    created_by_admin_id = Column(Integer, ForeignKey("admin_staff.admin_id"), nullable=False)
    trainer_id = Column(Integer, ForeignKey("trainer.trainer_id"), nullable=False)
    member_id = Column(Integer, ForeignKey("member.member_id"))

    # session has many-to-one relationship to Room
    room = relationship("Room", back_populates="sessions")
    # session has many-to-one relationship to AdminStaff (as creator)
    created_by_admin = relationship("Admin_staff", back_populates="sessions_created")
    # session has many-to-one relationship to Trainer
    trainer = relationship("Trainer", back_populates="sessions")
    # session has many-to-one (optional) relationship to Member
    member = relationship("Member", back_populates="sessions")
```

In summary, the role-specific service functions (member_service, trainer_service, admin_service) talk to the database only through this ORM layer using get_session(), .query(), .filter(), etc., so the report's "business logic" section and the ER/relational design stay tightly connected while also maintaining a high level or organization.