Health & Fitness Club Management System

Course: COMP 3005A

Sanvi Kaushik (101303354), Pardis Ehsani (101300400), Seham Khalifa (101295726)

Date: December 1, 2025

Demo Video: `https://vimeo.com/1142259951?share=copy&fl=sv&fe=ci`
GitHub Repository: `https://github.com/sanvikaushik/comp3005-fitness-orm`

1 Introduction

The **Health & Fitness Club Management System** is a full-stack application built using PostgreSQL, SQLAlchemy ORM, and Flask. It models end-to-end workflows for three personas: **Member**, **Trainer**, and **Administrative Staff**. The system implements all twelve required operations along with a trigger, view, and index, with all database access performed through SQLAlchemy ORM and validated through automated testing.

1.1 Team Contributions

- **Sanvi**
  - Led the development of the entire backend and implemented **nearly all functional code** for all three personas: Member, Trainer, and Administrative Staff.
  - Converted all initial team-provided CLI sketches into a complete **ORM-based backend** using SQLAlchemy.
  - Designed and implemented the full **ORM schema** (all models, relationships, constraints).
  - Built the complete **booking, scheduling, equipment, and payment pipelines**, ensuring all flows behaved consistently across roles.
  - Implemented the required **trigger, view, and index** directly in PostgreSQL.
  - Wrote all **pytest test suites** covering conflict checks, scheduling logic, billing, and dashboard loading.
  - Authored the complete project **README** and the full written report, **excluding Section 2**, which was drafted by Pardis.
  - Wrote all **normalization proofs and functional dependency analysis**.
  - Designed and wrote the **SQL commands** used in the pgAdmin and operations demo.
  - Integrated all features into the Flask application and ensured the entire system operated end-to-end.
- **Pardis**
  - Created the full **Entity–Relationship (ER) diagram**.
  - Authored the initial **functional requirements** and drafted **Section 2 (Requirements and ER Model Overview)** of the final report.
  - Wrote the **PlantUML code** used to generate the ER diagram.
  - Contributed to refining the ER structure and ensuring alignment with system requirements.
  - Assisted with validating the `README.md` setup instructions on a clean environment.

- **Seham**
  - Produced the detailed **ER-to-Relational Mapping** used to verify table structures and foreign key placement.
  - Wrote the **PlantUML mapping code** used to generate the relational mapping diagram.
  - Assisted in validating normalization decisions and ensuring the schema satisfied 2NF and 3NF.
  - Assisted with testing the `README.md` instructions from a fresh clone.

Demo Workflow Recap. The demonstration begins with an administrative login and database initialization via "Clear & Seed." It then proceeds through member workflows (profile creation with validation, health metrics, booking), trainer features (availability, classes, equipment), and administrative actions (rooms, scheduling, billing). The demo concludes with live verification of the trigger, view, index, ERD, and ORM mapping in pgAdmin.

2 Requirements and ER Model Overview

2.1 Functional Requirements (Role-Based)

The system supports three distinct user roles: **Member**, **Trainer**, and **Administrative Staff**. Each role has its own functional requirements and corresponding implemented operations.

Member Requirements

Members interact with the system through self-service functionality:

- Register with a unique email and provide basic profile information.
- Update personal profile details and fitness goals (e.g., target weight).
- Record health metrics such as weight, height, and heart rate as time-stamped history entries.
- Book or reschedule one-on-one private training sessions with trainers.
- Register for scheduled group fitness classes.

Implemented Member Operations:

[label=M1.]

1. User Registration
2. Profile Management
3. Health History Logging
4. Private Training Session Scheduling
5. Group Class Registration

Trainer Requirements

Trainers manage availability, schedules, and member interactions, with strict restrictions on data modification:

- Define availability time windows for private sessions and classes, ensuring no overlapping intervals.
- View their full schedule, including assigned private sessions and group classes.
- Look up members (case-sensitive search) and view member goals and most recent health metrics.
- Access member information in a read-only manner; trainers must *not* modify health metrics or profile fields.

Implemented Trainer Operations:

[label=T1.]

1. Set Availability
2. Schedule View
3. Member Lookup

Administrative Staff Requirements

Administrative staff supervise facility resources, scheduling, maintenance, and billing:

- Assign and manage room bookings for classes or private sessions, ensuring no double-booking conflicts.
- Log equipment issues, update maintenance statuses, and associate equipment with rooms as needed.
- Define new classes, assign trainers, allocate rooms, and update scheduling details.
- Generate billing items (e.g., charges for booked sessions), record payments, and track member account activity.
- Update operation statuses to reflect real-time system changes in the facility.

Implemented Administrative Staff Operations:

[label=A1.]

1. Room Bookings
2. Equipment Maintenance
3. Class Management
4. Billings and Payments

2.2 ER Model Specification

The ER model was developed using Gleek.io and captures all entities, relationships, and participation/cardinality rules used in the system.

Figure 1 shows the complete ER diagram used during conceptual design.

All entities in the model are **regular entities**; each has its own primary key and no weak entities are required. Associative entities such as `ClassRegistration` and `PrivateSession` include their own attributes and are modeled as full entities.

Entities and Attributes

Member: MemberID (PK), FirstName, LastName, Email, PhoneNumber, DateOfBirth, Gender.

Trainer: TrainerID (PK), FirstName, LastName, Email, PhoneNumber, Specialization, IsActive.

AdministrativeStaff: StaffID (PK), FirstName, LastName, Email, Password.

Room: RoomID (PK), RoomName, Capacity.

Equipment: EquipmentID (PK), RoomID (FK), Name, Status, LastServicedDate.

HealthMetric: MetricID (PK), MemberID (FK), RecordedAt, Weight, Height, TargetWeight, HeartRate.

ClassSchedule: ClassID (PK), ClassName, TrainerID (FK), RoomID (FK), DayOfWeek, StartTime, MaxCapacity.

ClassRegistration: RegistrationID (PK), MemberID (FK), ClassID (FK), DateRegistration.

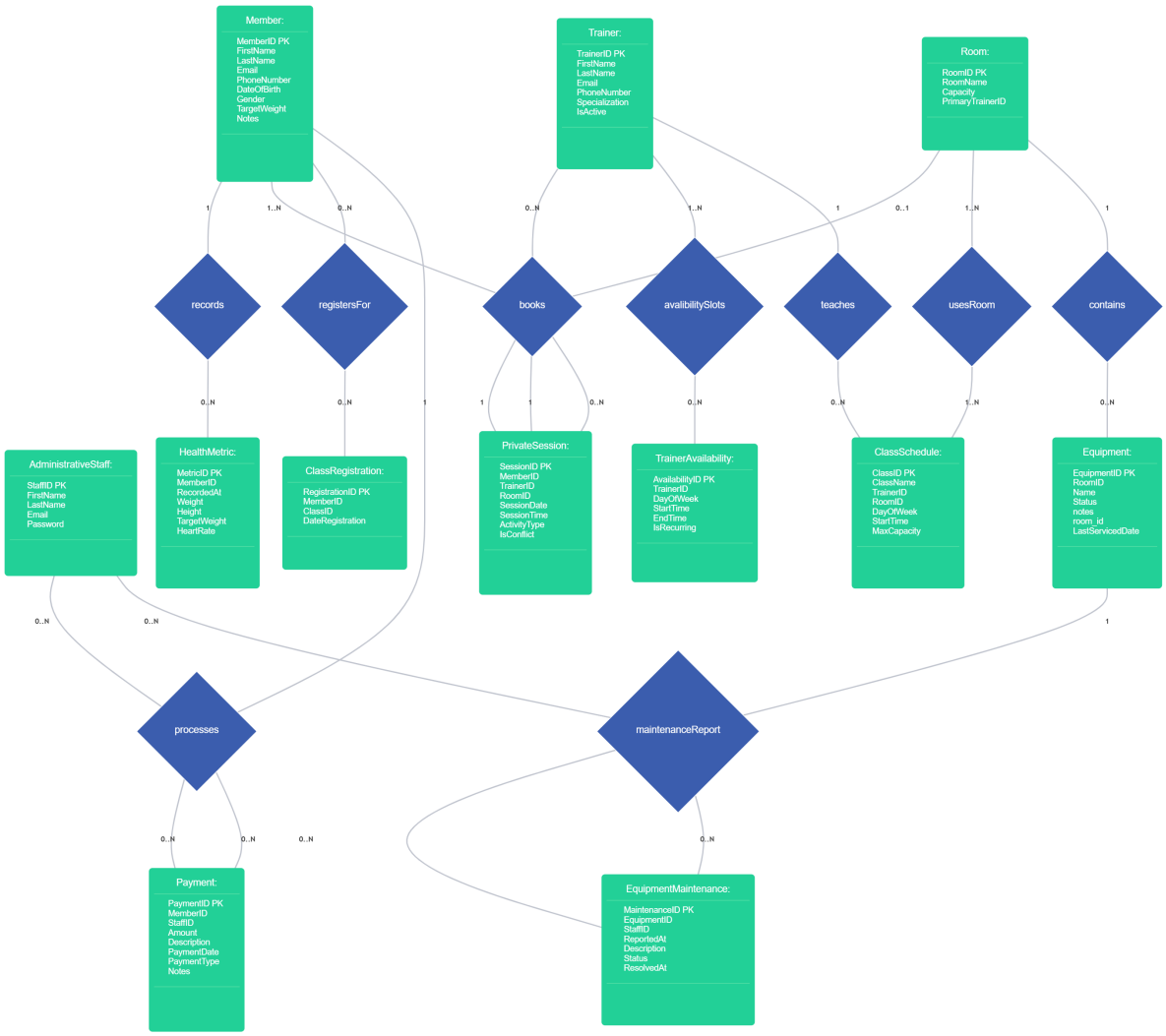

Figure 1: Entity–Relationship (ER) Diagram for the Health & Fitness Club System

PrivateSession: SessionID (PK), MemberID (FK), TrainerID (FK), RoomID (FK), SessionDate, SessionTime, ActivityType, IsConflict.

Payment: PaymentID (PK), MemberID (FK), StaffID (FK), Amount, PaymentDate, PaymentType, Notes.

TrainerAvailability: AvailabilityID (PK), TrainerID (FK), DayOfWeek, StartTime, EndTime, IsRecurring.

EquipmentMaintenance: MaintenanceID (PK), EquipmentID (FK), StaffID (FK), ReportedAt, Description, Status, ResolvedAt.

All of our entities are regular; each has its own primary key. `ClassRegistration` and `PrivateSession` are modeled as full associative entities with their own attributes (and PKs), not weak entities. `Equipment` belongs to a room via a foreign key, but is modeled as its own regular entity.

Named Relationships (Diamonds)

The ER diagram contains the following relationship types:

- **records** between Member and HealthMetric

- **books** between Member / Trainer / Room and PrivateSession
- **availabilitySlots** between Trainer and TrainerAvailability
- **teaches** between Trainer and ClassSchedule
- **usesRoom** between Room and ClassSchedule
- **processes** between AdministrativeStaff and Payment
- **contains** between Room and Equipment
- **maintenanceReport** between Equipment (and AdministrativeStaff) and EquipmentMaintenance
- **registersFor** between Member and ClassSchedule via ClassRegistration

2.3 Individual Linkages and Participation

This subsection captures the detailed semantics of each linkage, including participation (total vs. partial) and foreign key notes.

AdministrativeStaff–Payment (processes)

- AdministrativeStaff → Payment: **partial** participation. Not every administrative staff member processes payments.
- Payment → AdministrativeStaff: **partial** participation. Some payments may be automatic (e.g., online or Apple Pay) and not linked to staff.
- StaffID is a simple single-valued attribute; `Payment.StaffID` may be NULL.

Member–HealthMetric (records)

- Member –(1) records –(0..N) HealthMetric.
- HealthMetric → Member: **total** participation. Every `HealthMetric` must be associated with a Member.
- Member → HealthMetric: **partial** participation. A Member can have zero or many HealthMetrics.
- FK: `HealthMetric.MemberID` is NOT NULL.

Member–PrivateSession (books)

- Member –(1..N) books –(1..N) PrivateSession.
- Member participation: **partial**. A Member may not be in any private sessions.
- PrivateSession participation (on Member side): **total**. Every PrivateSession must be linked to at least one Member.

Member–ClassRegistration (registersFor)

- Member –(0..N) registersFor –(0..N) ClassRegistration.
- Member participation: **partial**. A Member can register in no classes.
- ClassRegistration participation (on Member side): **partial**. A class can temporarily have no participants.
- `ClassRegistration` is an associative entity with FKs `MemberID` and `ClassID`.

Member–Payment

- Member –(1..N) processes –(0..N) Payment.
- Member participation: **partial**. A Member can have no payments or some payments.
- Payment participation (on Member side): **total**. Every Payment must be associated with a Member.
- `Payment` is a regular entity with FKs `MemberID` and `StaffID`.

Trainer–PrivateSession (books)

- Trainer –(0..N) books –(1) PrivateSession.
- Trainer participation: **partial**. Not every trainer will have private sessions (they may only teach classes).
- PrivateSession participation (on Trainer side): **total**. Every PrivateSession must have a Trainer.

Trainer–TrainerAvailability (availabilitySlots)

- Trainer –(1) availabilitySlots –(0..N) TrainerAvailability.
- Trainer participation: **partial**. A Trainer might not always set explicit availability.
- TrainerAvailability participation: **total**. Every availability slot must belong to a Trainer.

Trainer–ClassSchedule (teaches)

- Trainer –(1) teaches –(0..N) ClassSchedule.
- ClassSchedule participation: **total**. Every ClassSchedule must have an assigned Trainer.
- Trainer participation: **partial**. A Trainer might not always teach classes.

Room–PrivateSession (books)

- Room –(0..N) books –(1..N) PrivateSession.
- Room participation: **partial**. A Room may not be booked for any PrivateSession (e.g., used only for classes).
- PrivateSession participation (on Room side): **partial**. A PrivateSession can have no physical room if it is held online.

Room–ClassSchedule (usesRoom)

- Room –(1..N) usesRoom –(1..N) ClassSchedule.
- Room participation: **partial**. A Room can exist without any scheduled classes.
- ClassSchedule participation: **total**. Every scheduled class must have a Room associated with it.

Room–Equipment (contains)

- Room –(1) contains –(0..N) Equipment.
- Equipment participation: **total**. Every piece of Equipment must have an assigned Room.
- Room participation: **partial**. Not every Room will contain Equipment, or a Room may contain many pieces of Equipment.

Equipment–EquipmentMaintenance (maintenanceReport)

- Equipment –(1) maintenanceReport –(0..N) EquipmentMaintenance.
- EquipmentMaintenance participation: **total**. Every maintenance report must be associated with an Equipment item.
- Equipment participation: **partial**. Not every piece of Equipment will have a maintenance report; it may be fully functional.
- FK: `EquipmentMaintenance.EquipmentID` and `EquipmentMaintenance.StaffID` are foreign keys.

AdministrativeStaff–EquipmentMaintenance

- AdministrativeStaff –(0..N) maintenanceReport –(0..N) EquipmentMaintenance.
- EquipmentMaintenance participation (on AdministrativeStaff side): **partial**. A maintenance report can be filed by someone else or by admin staff.
- AdministrativeStaff participation: **partial**. Not every admin staff member will be associated with maintenance reports.

2.4 Relationship Types and Cardinalities

Regular vs. Weak Entities

- **Regular Entity:** has its own primary key.
- **Weak Entity:** has no primary key of its own and depends on other relations.
- In this model, all entities are regular; there are no weak entities.
- `ClassRegistration` and `PrivateSession` are modeled as regular associative/relationship entities with their own attributes and primary keys.
- `Equipment` belongs to a Room (via FK) but is still modeled as a regular entity.

1:N Relationships (FK on the N side)

- Trainer (1) – (N) ClassSchedule → `ClassSchedule.TrainerID` is an FK.
- Room (1) – (N) ClassSchedule → `ClassSchedule.RoomID` is an FK.
- Trainer (1) – (N) TrainerAvailability → `TrainerAvailability.TrainerID` is an FK.
- Member (1) – (N) Payment → `Payment.MemberID` is an FK.
- Equipment (1) – (N) EquipmentMaintenance → `EquipmentMaintenance.EquipmentID` is an FK.
- AdministrativeStaff (1) – (N) Payment → `Payment.StaffID` is an FK.
- Room (1) – (N) Equipment → `Equipment.RoomID` is an FK.
- Trainer (1) – (N) PrivateSession → `PrivateSession.TrainerID` is an FK.

N:M Relationships (mapped via associative entities)

- Member – ClassSchedule mapped via `ClassRegistration`:

  $$\text{ClassRegistration}(\underline{\text{RegistrationID}}, \text{MemberID (FK)}, \text{ClassID (FK)}, \text{DateRegistration})$$

  A composite primary key (MemberID, ClassID) would also be possible, but a surrogate key is used instead.

- Member – Trainer mapped via `PrivateSession`, which also records Room, SessionDate, SessionTime, ActivityType, and IsConflict.

2.5 Attribute Types and FK Constraints

Attribute Types

- All attributes are **atomic** (cannot be decomposed further).
- **Simple attributes:** FirstName, LastName, Status, Gender, DateOfBirth, TrainerID, MemberID, etc.
- **Composite attributes:** N/A (none modeled).
- **Single-valued attributes:** All attributes in this schema are single-valued (e.g., each Member has one DateOfBirth, one Email).
- **Multi-valued attributes:** None. Potentially multi-valued concepts (e.g., health history, maintenance logs) are modeled as separate history tables, not as multi-valued attributes.

Foreign Key Constraints and Participation

- **Total participation** of an entity in a relationship is enforced by declaring the corresponding FK as `NOT NULL`.
- **Partial participation** is modeled by allowing the FK to be `NULL`.

2.6 Modeling Summary

In summary:

- All entities are regular, each with its own primary key.
- Relationship semantics, cardinalities, and participation (total vs. partial) are encoded via foreign keys and nullability.
- Associative entities `ClassRegistration` and `PrivateSession` capture N:M relationships and carry additional attributes.
- All attributes are atomic and single-valued, with no composite, multi-valued, or derived attributes at the ER level.

3 Database Design

3.1 Implementation View of the ER Model

The ER diagram in Figure 1 provides the conceptual foundation for the relational schema.

Figure 2 shows the ER-to-relational mapping used to convert conceptual relationships into foreign keys, associative entities, and surrogate-key primary keys.

This mapping preserves all cardinalities, participation constraints, and relationship semantics detailed in Figure 1.

The ER diagram (Figure 1) is fully realized in the relational schema through SQLAlchemy ORM. This subsection discusses implementation considerations rather than re-explaining conceptual details already covered earlier.

It reflects the requirements and participation/cardinality constraints detailed in Section *Requirements and ER Model Overview*. Therefore, this subsection focuses on how the ER model is realized in the relational schema and ORM.

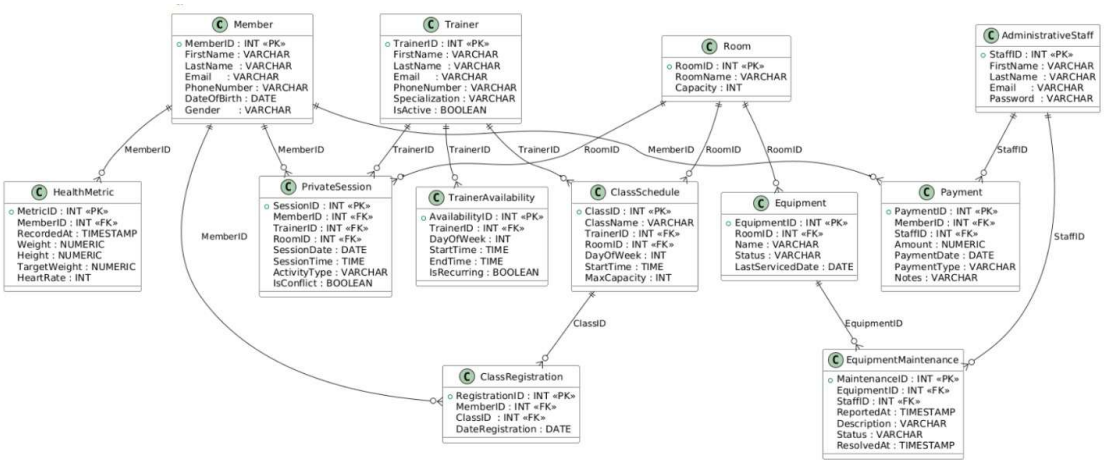

Figure 2: ER-to-Relational Mapping Diagram

Design Rationale (Implementation View).

- All entities from the ER model are implemented as regular tables with surrogate integer primary keys to align with SQLAlchemy's best practices.
- Associative relationships requiring attributes (`ClassRegistration`, `PrivateSession`) are implemented as full entities rather than weak entities, ensuring natural mapping to ORM classes.
- Total/partial participation rules from the ER model are enforced through:
  - `nullable=False` constraints on mandatory foreign keys;
  - cascading deletes for historical tables;
  - service-layer validations for temporal and booking constraints.
- Multi-valued attributes do not appear in this schema; concepts that could be multi-valued (such as a member's historical health metrics or equipment maintenance history) are modeled using separate history tables rather than repeating columns, ensuring 3NF compliance.

3.2 ER to Relational Mapping

The conceptual ER elements map to the relational schema as follows (without repeating the previously listed attributes):

- Each ER entity becomes a table: `member`, `trainer`, `room`, `equipment`, `health_metric`, `class_schedule`, `class_registration`, `private_session`, `trainer_availability`, `equipment_maintenance`, `payment`.

- All identifying relationships are implemented using foreign keys on the "many" side.
- The only M:N relationship (Member–ClassSchedule) is implemented by the associative table `class_registration`.
- All relationship semantics from the ER model (1:N, M:N, total/partial participation) are enforced through schema constraints and ORM logic, not re-described here.

ORM Alignment. SQLAlchemy's declarative mapping embeds:

- `relationship()` attributes matching all ER relationships;
- eager and lazy loading strategies depending on dashboard vs. lookup use-cases;
- cascading rules such as `"all, delete-orphan"` for history tables;
- uniqueness constraints (e.g., `member.email`) to enforce ER keys.

This provides a true one-to-one correspondence between the ER model and the implemented schema without repeating conceptual relationship descriptions already provided earlier.

3.3 Normalization and Functional Dependencies

This section shows, using functional dependencies (FDs), that all relations in the Health & Fitness Club schema are in **Third Normal Form (3NF)**. We follow the lecture convention:

- A relation R is in **1NF** if all attributes are atomic (no multi-valued or composite attributes) and every tuple contains only single, indivisible values.
- A relation R is in **2NF** if it is in 1NF and no non–key attribute is partially dependent on a *proper subset* of any candidate key.
- A relation R is in **3NF** if for every non–trivial FD $X \to A$ on R, either (i) X is a superkey of R, or (ii) A is part of some candidate key.

Throughout, we use the following notation:

$$R(\text{attributes}) \quad \text{with FDs} \quad F$$

Global observation (1NF). Every relation uses atomic attributes only (no arrays, repeating groups, or composite fields), and historical data is modeled with separate tables (`HealthMetric`, `EquipmentMaintenance`) instead of multi-valued columns. Therefore, *all* relations are in 1NF.

Global observation (2NF). Each relation in our design uses a *single-attribute surrogate primary key* (`MemberID`, `TrainerID`, `MetricID`, `ClassID`, `RegistrationID`, `SessionID`, `PaymentID`, etc.). Since no relation has a composite primary key, no non-key attribute can depend on a proper subset of a key. Hence, every relation is automatically in 2NF.

Below we list the key functional dependencies for each relation and verify 3NF.

Member

Member(<u>MemberID</u>, FirstName, LastName, Email, PhoneNumber, DateOfBirth, Gender, TargetWeight, Notes)

Assumed FDs:

$$\text{MemberID} \to \text{FirstName, LastName, Email, PhoneNumber, DateOfBirth, Gender, TargetWeight, Notes}$$
$$\text{Email} \to \text{MemberID, FirstName, LastName, } \ldots$$

Candidate keys: {MemberID} (primary), and (by the UNIQUE constraint) {Email}. For each non-trivial FD, the left side is a key (or superkey), so the 3NF condition holds. There are no FDs where a non-key attribute determines another non-key attribute (for example, `PhoneNumber` does not determine `TargetWeight`). Therefore, **Member is in 3NF**.

Trainer

$$\text{Trainer}(\underline{\text{TrainerID}}, \text{FirstName}, \text{LastName}, \text{Email}, \text{PhoneNumber}, \text{Specialization}, \text{IsActive})$$

FDs:

$$\text{TrainerID} \rightarrow \text{FirstName}, \text{LastName}, \text{Email}, \text{PhoneNumber}, \text{Specialization}, \text{IsActive}$$

Candidate key: {TrainerID}. All non-key attributes depend directly and only on the key; no non-key attribute determines another non-key attribute. Hence, **Trainer is in 3NF**.

AdministrativeStaff

$$\text{AdministrativeStaff}(\underline{\text{StaffID}}, \text{FirstName}, \text{LastName}, \text{Email}, \text{Password})$$

FDs:

$$\text{StaffID} \rightarrow \text{FirstName}, \text{LastName}, \text{Email}, \text{Password}$$

Candidate key: {StaffID}. As above, the only non-trivial FDs have a key on the left; **AdministrativeStaff is in 3NF**.

Room

$$\text{Room}(\underline{\text{RoomID}}, \text{RoomName}, \text{Capacity})$$

FDs:

$$\text{RoomID} \rightarrow \text{RoomName}, \text{Capacity}$$

Candidate key: {RoomID}. No non-key attribute determines another non-key attribute, so **Room is in 3NF**.

Equipment

$$\text{Equipment}(\underline{\text{EquipmentID}}, \text{RoomID}, \text{Name}, \text{Status}, \text{LastServicedDate})$$

FDs:

$$\text{EquipmentID} \rightarrow \text{RoomID}, \text{Name}, \text{Status}, \text{LastServicedDate}$$

Candidate key: {EquipmentID}. Attributes such as `Status` and `LastServicedDate` do not functionally determine each other within this relation; all depend on the key only. **Equipment is in 3NF**.

HealthMetric

$$\text{HealthMetric}(\underline{\text{MetricID}}, \text{MemberID}, \text{RecordedAt}, \text{Weight}, \text{Height}, \text{TargetWeight}, \text{HeartRate})$$

FDs:

$$\text{MetricID} \rightarrow \text{MemberID}, \text{RecordedAt}, \text{Weight}, \text{Height}, \text{TargetWeight}, \text{HeartRate}$$

(Optionally, we may also assume (MemberID, RecordedAt) is a candidate key for business reasons, but we keep the surrogate key `MetricID` as primary.)

In either case, all non-key attributes are determined directly by the key, and there is no chain like MemberID $\rightarrow$ Weight $\rightarrow$ HeartRate. Thus, **HealthMetric is in 3NF**.

ClassSchedule

$$\text{ClassSchedule}(\underline{\text{ClassID}}, \text{ClassName}, \text{TrainerID}, \text{RoomID}, \text{DayOfWeek}, \text{StartTime}, \text{MaxCapacity})$$

FDs:

$$\text{ClassID} \rightarrow \text{ClassName}, \text{TrainerID}, \text{RoomID}, \text{DayOfWeek}, \text{StartTime}, \text{MaxCapacity}$$

Candidate key: {ClassID}. There is no dependency such as RoomID $\rightarrow$ MaxCapacity inside this relation; capacity belongs to the class instance, not the room. Hence, **ClassSchedule is in 3NF**.

ClassRegistration

$$\text{ClassRegistration}(\underline{\text{RegistrationID}}, \text{MemberID}, \text{ClassID}, \text{DateRegistration})$$

Natural FDs:

$$\text{RegistrationID} \rightarrow \text{MemberID}, \text{ClassID}, \text{DateRegistration}$$
$$(\text{MemberID}, \text{ClassID}) \rightarrow \text{DateRegistration}$$

Candidate keys: {RegistrationID} (surrogate key) and {MemberID, ClassID} (natural key). For both keys, every non-key attribute is fully dependent on the whole key, and there is no transitive dependency between non-key attributes. Therefore, **ClassRegistration is in 3NF**.

PrivateSession

$$\text{PrivateSession}(\underline{\text{SessionID}}, \text{MemberID}, \text{TrainerID}, \text{RoomID}, \text{SessionDate}, \text{SessionTime}, \text{ActivityType}, \text{IsConflict})$$

FD:

$$\text{SessionID} \rightarrow \text{MemberID}, \text{TrainerID}, \text{RoomID}, \text{SessionDate}, \text{SessionTime}, \text{ActivityType}, \text{IsConflict}$$

Candidate key: {SessionID}. Once the session is identified, all other attributes are fixed. None of the non-key attributes determines any other non-key attribute inside this relation (for example, `ActivityType` does not determine `RoomID`). Thus, **PrivateSession is in 3NF**.

TrainerAvailability

$$\text{TrainerAvailability}(\underline{\text{AvailabilityID}}, \text{TrainerID}, \text{DayOfWeek}, \text{StartTime}, \text{EndTime}, \text{IsRecurring})$$

FD:

$$\text{AvailabilityID} \rightarrow \text{TrainerID}, \text{DayOfWeek}, \text{StartTime}, \text{EndTime}, \text{IsRecurring}$$

Candidate key: {AvailabilityID}. All other attributes depend only on this key; no non-key attribute determines another. Hence, **TrainerAvailability is in 3NF**.

EquipmentMaintenance

$$\text{EquipmentMaintenance}(\underline{\text{MaintenanceID}}, \text{EquipmentID}, \text{StaffID}, \text{ReportedAt}, \text{Description}, \text{Status}, \text{ResolvedAt})$$

FD:

$$\text{MaintenanceID} \rightarrow \text{EquipmentID}, \text{StaffID}, \text{ReportedAt}, \text{Description}, \text{Status}, \text{ResolvedAt}$$

Candidate key: {MaintenanceID}. There is no FD such as Status → ResolvedAt enforced at the schema level; resolved time is a separate attribute. Therefore, **EquipmentMaintenance is in 3NF**.

Payment

$$\text{Payment}(\underline{\text{PaymentID}}, \text{MemberID}, \text{StaffID}, \text{Amount}, \text{PaymentDate}, \text{PaymentType}, \text{Notes})$$

FD:

$$\text{PaymentID} \rightarrow \text{MemberID}, \text{StaffID}, \text{Amount}, \text{PaymentDate}, \text{PaymentType}, \text{Notes}$$

Candidate key: {PaymentID}. Again, no non-key attribute determines another non-key attribute; all depend directly on the key. Thus, **Payment is in 3NF**.

Summary. All relations are in 1NF (atomic attributes), 2NF (no composite keys, so no partial dependencies), and 3NF (for every non-trivial FD, the left side is a superkey, and there are no non-key → non-key dependencies). Therefore, the overall schema is **fully normalized to 3NF**.

4 Implementation

4.1 Technology Stack

- Flask web application (`app/web_app.py`)
- SQLAlchemy ORM for schema creation and CRUD
- PostgreSQL with pgAdmin for validation
- Python scripts for clearing/seeding
- Pytest for validation (`tests/`)

4.2 ORM Usage

SQLAlchemy ORM abstracts SQL by mapping tables to Python classes and performing queries through session objects. Lazy loading is used for simple record fetches; eager-loading strategies (`joinedload`, `selectinload`) support dashboards and prevent `DetachedInstanceError`.

4.2.1 Example ORM Models

Listing 1: Member and HealthMetric Models

```python
class Member(Base):
    __tablename__ = "member"
    member_id = mapped_column(Integer, primary_key=True)
    email = mapped_column(String(255), nullable=False, unique=True)
    health_metrics = relationship(
        "HealthMetric",
        back_populates="member",
        cascade="all, delete-orphan",
        order_by="HealthMetric.timestamp.desc()"
    )

class HealthMetric(Base):
    __tablename__ = "health_metric"
    metric_id = mapped_column(Integer, primary_key=True)
    member_id = mapped_column(ForeignKey("member.member_id"), nullable=False)
    weight = mapped_column(Numeric(5,2), nullable=True)
    heart_rate = mapped_column(Integer, nullable=True)
    timestamp = mapped_column(
        TIMESTAMP(timezone=False),
        server_default=text("NOW()")
    )
    member = relationship("Member", back_populates="health_metrics")
```

4.2.2 Example Service Query

Listing 2: Health Metric Query

```python
def get_health_history(session, member_id):
    stmt = (
        select(HealthMetric)
        .where(HealthMetric.member_id == member_id)
        .order_by(HealthMetric.timestamp.desc())
    )
    return list(session.scalars(stmt))
```

4.2.3 Lazy vs. Eager Loading in Practice

For simple lookups (e.g., booking a class or recording a payment), the system uses **lazy loading** via `session.get(...)` to fetch a single object by primary key:

Listing 3: Lazy Loading for Targeted Lookups

```
def register_for_class(session, *, member_id: int, class_id: int):
    member = session.get(Member, member_id)          # lazy load
    if not member:
        raise ValueError("Member not found")

    cls = session.get(ClassSchedule, class_id)       # lazy load
    if not cls:
        raise ValueError("Class not found")
    ...
```

Here, only the `Member` and `ClassSchedule` objects are needed to create a `ClassRegistration` and `BillingItem`, so loading related collections eagerly would be wasteful.

In contrast, dashboard and schedule views require a full graph of related objects (member profile, trainer, room, sessions, and metrics). These use **eager loading** to avoid N+1 queries and `DetachedInstanceError`:

Listing 4: Eager Loading for Member Dashboard

```
def get_member_dashboard(session, member_id: int, now: datetime | None = None)
    -> dict:
    stmt = (
        select(Member)
        .options(
            joinedload(Member.class_registrations)
                .joinedload(ClassRegistration.class_schedule)
                .joinedload(ClassSchedule.trainer),
            joinedload(Member.class_registrations)
                .joinedload(ClassRegistration.class_schedule)
                .joinedload(ClassSchedule.room),
            joinedload(Member.private_sessions)
                .joinedload(PrivateSession.trainer),
            joinedload(Member.private_sessions)
                .joinedload(PrivateSession.room),
        )
        .where(Member.member_id == member_id)
    )
    member = session.execute(stmt).unique().scalars().first()
    ...
```

Similarly, the trainer schedule eagerly preloads all related sessions, classes, rooms, and billing items:

Listing 5: Eager Loading for Trainer Schedule

```
trainer_stmt = (
    select(Trainer)
    .options(
        joinedload(Trainer.private_sessions)
            .joinedload(PrivateSession.member),
        joinedload(Trainer.private_sessions)
            .joinedload(PrivateSession.room),
        joinedload(Trainer.classes)
            .joinedload(ClassSchedule.room),
        joinedload(Trainer.classes)
            .selectinload(ClassSchedule.registrations),
    )
    .where(Trainer.trainer_id == trainer_id)
)
trainer = session.scalars(trainer_stmt).first()
```

Lazy loading keeps simple operations efficient and focused; eager loading ensures that UI-heavy views (member dashboard and trainer schedule) have all related objects available with a small number of SQL queries.

4.3 Database Features (Trigger, View, Index)

Trigger: `trg_update_member_last_metric` updates `member.last_metric_at` whenever a new `HealthMetric` is inserted.

View: `member_latest_metric_view` returns each member's most recent health metric by joining `member` and `health_metric`.

Index: `idx_health_metric_member_id` accelerates member history lookups in `get_health_history`.

5 Role-Based Features — 12 Operations

5.1 Member Operations

5.1.1 Operation M1: Create/Update Member

Listing 6: Create Member with Uniqueness Enforcement

```
def create_member(session: Session, *, email: str, target_weight: float | None =
    None, ...):
    member = Member(
        email=email,
        target_weight=_normalize_target_weight(target_weight),
        ...
    )
    session.add(member)
    try:
        session.commit()
    except IntegrityError:
        session.rollback()
        raise ValueError("Email or phone already exists")
    session.refresh(member)
    return member
```

The `email` column is declared as `unique=True` at the ORM level, and the service layer catches `IntegrityError` to prevent duplicate emails across different members.

5.1.2 Operation M2: Log Health Metrics

Listing 7: Append-Only Health Metrics

```
def log_health_metric(session: Session, member_id: int,
                      weight: float | None = None,
                      heart_rate: int | None = None) -> HealthMetric:
    metric = HealthMetric(
        member_id=member_id,
        weight=weight,
        heart_rate=heart_rate,
    )
    session.add(metric)
    session.commit()
    session.refresh(metric)
    return metric
```

Health metrics are append-only; each call inserts a new `HealthMetric` object. Queries such as `get_health_history` and the `member_latest_metric_view` rely on ordering by `timestamp` rather than overwriting previous records.

5.1.3 Operation M3: Book Private Session (Conflict Checks)

Listing 8: Private Session Booking Conflicts

```
overlap_ps = and_(PrivateSession.start_time < end_time,
                  PrivateSession.end_time > start_time)
overlap_cls = and_(ClassSchedule.start_time < end_time,
                   ClassSchedule.end_time > start_time)

# Room conflicts with other private sessions
room_session_stmt = select(PrivateSession).where(
    PrivateSession.room_id == room_id,
    overlap_ps,
)
if session.scalars(room_session_stmt).first():
    raise ValueError("Room is already booked for another private session")

# Room conflicts with classes
room_class_stmt = select(ClassSchedule).where(
    ClassSchedule.room_id == room_id,
    overlap_cls,
)
if session.scalars(room_class_stmt).first():
    raise ValueError("Room is already booked for a class in that time")

# Trainer conflicts
trainer_session_stmt = select(PrivateSession).where(
    PrivateSession.trainer_id == trainer_id,
    overlap_ps,
)
if session.scalars(trainer_session_stmt).first():
    raise ValueError("Trainer is already booked for another private session")

trainer_class_stmt = select(ClassSchedule).where(
    ClassSchedule.trainer_id == trainer_id,
    overlap_cls,
)
if session.scalars(trainer_class_stmt).first():
    raise ValueError("Trainer is already teaching a class in that time")
```

These checks ensure that no member, trainer, or room is double-booked for overlapping time windows when creating a `PrivateSession`.

5.1.4 Operation M4: Register for Group Class (Capacity and Duplicates)

Listing 9: Group Class Registration Constraints

```
# Prevent duplicate registration
exists_stmt = select(ClassRegistration).where(
    ClassRegistration.member_id == member_id,
    ClassRegistration.class_id == class_id,
)
if session.scalars(exists_stmt).first():
    raise ValueError("Member already registered for this class")

# Capacity check
count_stmt = (
    select(func.count(ClassRegistration.registration_id))
    .where(ClassRegistration.class_id == class_id)
)
current_count = session.scalar(count_stmt) or 0
if current_count >= cls.capacity:
```

```
    raise ValueError("Class is full")
```

This prevents duplicate registrations for the same member/class pair and enforces the class capacity constraint.

5.2 Trainer Operations

5.2.1 Operation T1: View Schedule (Eager Loading)

Listing 10: Trainer Schedule with Eager Loading

```
trainer_stmt = (
    select(Trainer)
    .options(
        joinedload(Trainer.private_sessions)
            .joinedload(PrivateSession.member),
        joinedload(Trainer.private_sessions)
            .joinedload(PrivateSession.room),
        joinedload(Trainer.classes)
            .joinedload(ClassSchedule.room),
        joinedload(Trainer.classes)
            .selectinload(ClassSchedule.registrations),
        joinedload(Trainer.availabilities),
    )
    .where(Trainer.trainer_id == trainer_id)
)
trainer = session.scalars(trainer_stmt).first()
```

The schedule view uses eager loading so that all related objects (private sessions, group classes, members, rooms, and availability windows) are available without additional queries when rendering the trainer dashboard.

5.2.2 Operation T3: Set Availability (Trainer-Time Conflicts)

Listing 11: Trainer Availability Overlap Check

```
overlap_stmt = select(TrainerAvailability).where(
    TrainerAvailability.trainer_id == trainer_id,
    TrainerAvailability.day_of_week == day_of_week,
    or_(
        and_(TrainerAvailability.start_time <= start,
             TrainerAvailability.end_time > start),
        and_(TrainerAvailability.start_time < end,
             TrainerAvailability.end_time >= end),
        and_(TrainerAvailability.start_time >= start,
             TrainerAvailability.end_time <= end),
    ),
)
if session.scalars(overlap_stmt).first():
    raise ValueError("Availability window overlaps with an existing one")
```

This logic guarantees that trainers cannot define overlapping availability windows for the same day.

5.3 Additional Admin Operations

5.3.1 Operation A2: Room Booking Adjustments (Reuse of Conflict Logic)

Listing 12: Admin Reschedule Using Member Logic

```
def admin_reassign_session_room(session: Session, *,
                                session_id: int,
```

```
                                    new_room_id: int,
                                    new_start: datetime | None = None,
                                    new_end: datetime | None = None) ->
                                        PrivateSession:
    session_obj = session.get(PrivateSession, session_id)
    if not session_obj:
        raise ValueError("Private session not found")

    room = session.get(Room, new_room_id)
    if not room:
        raise ValueError("Room not found")

    # Reuse member-side conflict checks
    updated = reschedule_private_session(
        session,
        session_id=session_id,
        new_room_id=new_room_id,
        new_start=new_start,
        new_end=new_end,
    )
    return updated
```

The admin rescheduling operation delegates to `reschedule_private_session`, which applies the same room/trainer/member conflict rules as initial booking, ensuring consistent behavior across roles.

5.3.2 Operation A3: Class Management (Group Session Conflicts)

Listing 13: Admin Class Reschedule Using Trainer Logic

```
def admin_reschedule_class(session: Session, *,
                           class_id: int,
                           new_room_id: int,
                           new_start: datetime,
                           new_end: datetime) -> ClassSchedule:
    cls = session.get(ClassSchedule, class_id)
    if not cls:
        raise ValueError("Class not found")

    room = session.get(Room, new_room_id)
    if not room:
        raise ValueError("Room not found")

    updated = create_or_update_class(
        session,
        trainer_id=cls.trainer_id,
        room_id=new_room_id,
        name=cls.name,
        capacity=cls.capacity,
        start_time=new_start,
        end_time=new_end,
        price=float(cls.price or 0),
        class_id=cls.class_id,
    )
    return updated
```

`create_or_update_class` enforces room and trainer conflict checks for group classes, so admins and trainers share the same collision rules for class windows.

5.3.3 Operation A5: Billing and Payments

Listing 14: Recording Payments via ORM

```python
def record_payment(session: Session,
                   member_id: int,
                   amount: float,
                   description: str | None = None,
                   private_session_id: int | None = None) -> Payment:
    member = session.get(Member, member_id)  # lazy load
    if not member:
        raise ValueError(f"Member {member_id} not found.")

    payment = Payment(
        member_id=member_id,
        amount=amount,
        description=description,
        paid_at=datetime.utcnow(),
        private_session_id=private_session_id,
    )
    session.add(payment)
    session.commit()
    session.refresh(payment)
    return payment
```

Payment recording uses lazy loading to fetch the target member, then inserts a `Payment` object. The updated payment state is later surfaced in the member dashboard and trainer schedule via the eager-loaded queries described above.

6 Testing

Pytest suites simulate all conflict scenarios:

- Trainer availability overlaps
- Member/trainer/room double-booking
- Class capacity enforcement
- Equipment ID validation
- Numeric validation for health metrics

Smoke tests confirm initialization (trigger, view, index), eager-loading, and cross-dashboard billing updates.

7 Challenges and Edge Cases

The main difficulties in the project all came from keeping the ORM, scheduling logic, and database features in sync. This section briefly explains each challenge and points to the concrete lines where it was addressed.

Avoiding `DetachedInstanceError` (lazy vs. eager loading)

Early versions of the dashboard loaded a `Member` and then tried to access related collections after the session had been closed, causing `DetachedInstanceError`. The fix was to explicitly eager-load all relationships needed by the view:

Listing 15: Eager-loading for Member Dashboard

```python
member_stmt = (
    select(Member)
    .options(
        joinedload(Member.class_registrations)
```

```
            .joinedload(ClassRegistration.class_schedule)
            .joinedload(ClassSchedule.trainer),
        joinedload(Member.class_registrations)
            .joinedload(ClassRegistration.class_schedule)
            .joinedload(ClassSchedule.room),
        joinedload(Member.private_sessions)
            .joinedload(PrivateSession.trainer),
        joinedload(Member.private_sessions)
            .joinedload(PrivateSession.room),
    )
    .where(Member.member_id == member_id)
)
member = session.execute(member_stmt).unique().scalars().first()
```

The same pattern is used in `get_trainer_schedule` with `joinedload`/`selectinload` to ensure all sessions, classes, and rooms are available while the session is still open.

Shared booking conflict logic across roles

Another challenge was making sure that Members, Trainers, and Admins all use the *same* conflict rules for rooms, trainers, and members when booking or rescheduling sessions.

For member-side private session booking, conflicts are checked centrally in `book_private_session`:

Listing 16: Room and Trainer Conflict Checks

```
overlap_ps = and_(PrivateSession.start_time < end_time,
                  PrivateSession.end_time > start_time)
overlap_cls = and_(ClassSchedule.start_time < end_time,
                   ClassSchedule.end_time > start_time)

room_session_stmt = select(PrivateSession).where(
    PrivateSession.room_id == room_id,
    overlap_ps,
)
if session.scalars(room_session_stmt).first():
    raise ValueError("Room is already booked for another private session")

trainer_class_stmt = select(ClassSchedule).where(
    ClassSchedule.trainer_id == trainer_id,
    overlap_cls,
)
if session.scalars(trainer_class_stmt).first():
    raise ValueError("Trainer is already teaching a class in that time")
```

Admin adjustments reuse the same logic instead of re-implementing it. For example, `admin_reassign_session_room` delegates directly to `reschedule_private_session`:

Listing 17: Admin Reuses Member Reschedule Logic

```
updated = reschedule_private_session(
    session,
    session_id=session_id,
    new_room_id=new_room_id,
    new_start=new_start,
    new_end=new_end,
)
```

Similarly, `admin_reschedule_class` wraps `create_or_update_class`, so trainer/room conflicts for group classes are enforced uniformly for both trainers and admins.

Cascading billing updates across dashboards

Because billing involves both charges and payments, the challenge was to keep everything in sync when a member books, reschedules, or pays for a session.

For private sessions, the helper `_ensure_private_billing` guarantees that each `PrivateSession` has a matching `BillingItem`, and that it is updated if the session changes:

Listing 18: Ensuring Private Session Billing

```
bill = session.scalar(
    select(BillingItem).where(
        BillingItem.private_session_id == private_session.session_id
    )
)
if bill:
    bill.description = desc
    bill.amount = float(private_session.price or 0)
    bill.updated_at = datetime.utcnow()
    ...
else:
    bill = BillingItem(
        member_id=private_session.member_id,
        private_session_id=private_session.session_id,
        trainer_id=trainer.trainer_id if trainer else None,
        amount=float(private_session.price or 0),
        description=desc,
        status="pending",
    )
    session.add(bill)
```

Admin-side payments are recorded through:

Listing 19: Recording a Payment

```
payment = Payment(
    member_id=member_id,
    amount=amount,
    description=description,
    paid_at=datetime.utcnow(),
    private_session_id=private_session_id,
)
session.add(payment)
session.commit()
```

The member dashboard and trainer schedule then surface these changes by eager-loading `BillingItem` and `Payment` objects in their queries, so updates made in the admin UI automatically appear in member and trainer views without additional SQL.

Integrating trigger, view, and index with ORM

Finally, the trigger, view, and index needed to coexist cleanly with SQLAlchemy's metadata. The solution was to let SQLAlchemy create all base tables, then apply the extra DDL explicitly in `init_db()`:

Listing 20: Trigger, View, and Index Setup

```
Base.metadata.create_all(bind=engine)

with engine.begin() as conn:
    conn.execute(text("""
        ALTER TABLE member
            ADD COLUMN IF NOT EXISTS last_metric_at TIMESTAMP;
    """))

    conn.execute(text("""
```

```
        CREATE OR REPLACE FUNCTION update_last_metric() RETURNS TRIGGER AS $$
        BEGIN
            UPDATE member
            SET last_metric_at = NEW.timestamp
            WHERE member_id = NEW.member_id;
            RETURN NEW;
        END;
        $$ LANGUAGE plpgsql;
    """))

    conn.execute(text("""
        CREATE TRIGGER trg_update_member_last_metric
        AFTER INSERT ON health_metric
        FOR EACH ROW
        EXECUTE FUNCTION update_last_metric();
    """))

    conn.execute(text("""
        CREATE OR REPLACE VIEW member_latest_metric_view AS
        SELECT ...
    """))

    conn.execute(text("""
        CREATE INDEX IF NOT EXISTS idx_health_metric_member_id
        ON health_metric(member_id);
    """))
```

Keeping all non-ORM DDL inside `init_db()` made it easy to re-run schema setup during tests and ensured that the trigger, view, and index remained compatible with the ORM mappings.

8 Conclusion

The system delivers a cohesive ORM-first architecture, validated through the Flask UI, pgAdmin, and automated tests. All requirements; twelve operations, one trigger, one view, one index, and full ERD/relational mapping, are fully satisfied. Future enhancements include stronger auth, richer analytics, and multi-room optimization.