DATABASE FOUNDATION PROJECT REPORT

Project contributor: Suppapoo Ekpipattana

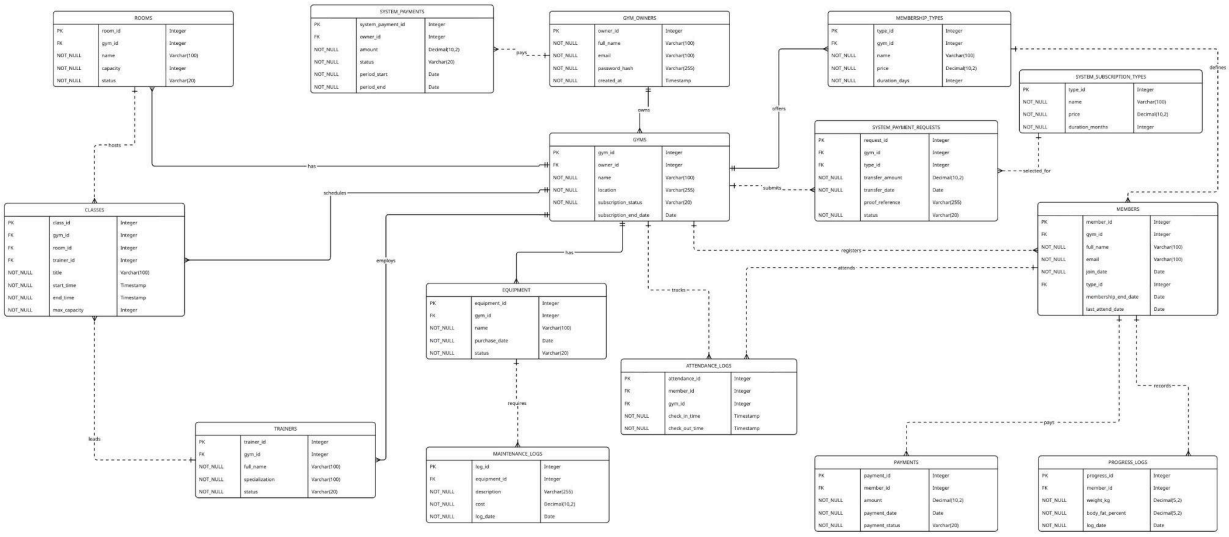
Database Engine: PostgreSQL (PL/pgSQL)

Concern: Every UI here is AI generated.

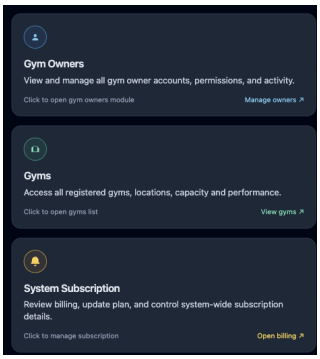
The Gym Platform is designed to manage the operations of multiple independent fitness centers within a single software ecosystem.

The system relies on PL/pgSQL Functions to handle logic and some logic is implemented in the backend.

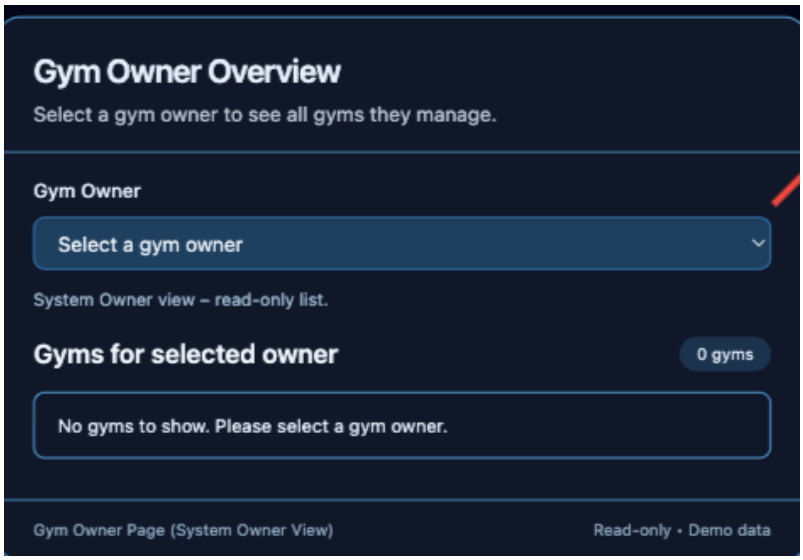
ER diagram:



The Super Admin (System Owner) :



- `Gym_owners` Info(Customers).

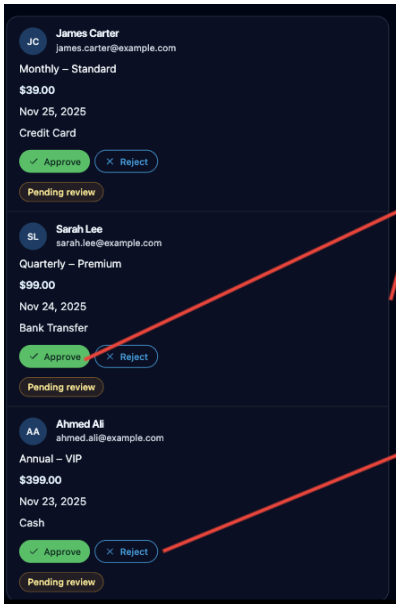



```
create function get_owners_gyms(_owner_id integer)
    returns TABLE(r_gym_id integer, r_name character varying, r_location character varying, r_status character varying)
    language plpgsql
as
$$
BEGIN
    RETURN QUERY
    SELECT
        gym_id,
        name,
        location,
        subscription_status
    FROM gyms
    WHERE owner_id = _owner_id
    ORDER BY gym_id ASC;
END;
$$;

alter function get_owners_gyms(integer) owner to root;
```

2	Muscle Factory BKK	Bangkok	Active
3	Power Gym	Bangkok	Active

This will show all the gyms the link to that owner id

- Handles `system_payments` (from Owners to Admin).




```
create function approve_payment_request(_request_id integer) returns text
    _type_id INT;
    _status VARCHAR;
    _result_msg TEXT;
BEGIN
    SELECT gym_id, type_id, status
    INTO _gym_id, _type_id, _status
    FROM system_payment_requests
    WHERE request_id = _request_id;

    IF _gym_id IS NULL THEN
        RETURN 'Error: Request ID not found.';
    END IF;

    IF _status = 'Approved' THEN
        RETURN 'Error: This request is already approved.';
    END IF;

    SELECT make_system_payment( _gym_id _gym_id,  _plan_type_id _type_id) INTO _result_msg;

    UPDATE system_payment_requests
    SET status = 'Approved'
    WHERE request_id = _request_id;

    RETURN 'Approved: ' || _result_msg;
```

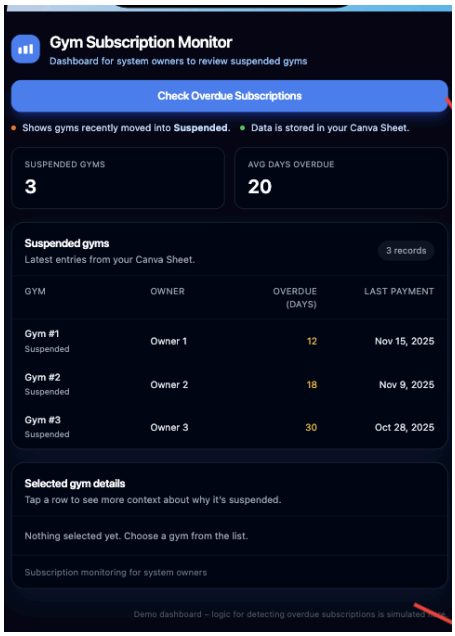
```
create function reject_payment_request(_request_id integer, _reason text) returns text
    SELECT gym_id, status
    INTO _gym_id, _status
    FROM system_payment_requests
    WHERE request_id = _request_id;

    IF _gym_id IS NULL THEN
        RETURN 'Error: Request ID not found.';
    END IF;

    IF _status != 'Pending' THEN
        RETURN 'Error: Cannot reject. Status is already ' || _status;
    END IF;

    UPDATE system_payment_requests
    SET status = 'Rejected',
        admin_note = _reason
    WHERE request_id = _request_id;
```

```
create function make_system_payment(_gym_id integer, _plan_type_id integer) returns text
    IF _amount IS NULL THEN
        RETURN 'Error: Subscription Plan ID ' || _plan_type_id || ' not found.';
    END IF;

    SELECT COALESCE(subscription_end_date, CURRENT_DATE)
    INTO _start_date
    FROM gyms
    WHERE gym_id = _gym_id;

    IF _start_date < CURRENT_DATE THEN
        _start_date := CURRENT_DATE;
    END IF;

    _end_date := _start_date + (_months || ' month')::INTERVAL;

    INSERT INTO system_payments (owner_id, gym_id, type_id, amount, status, period_start, period_end)
    VALUES ( owner_id _owner_id,  gym_id _gym_id,  type_id _plan_type_id,  amount _amount,  status 'Paid',  perio
    RETURNING system_payment_id INTO _payment_id;

    UPDATE gyms
    SET subscription_status = 'Active',
        subscription_end_date = _end_date
    WHERE gym_id = _gym_id;
```

After click approve it will run make payment system to renew subscription.

When the gym owner submitted the payment. Admin need to manually approve or reject

- Controls `Gym_status_payment_overdue`




```
create function check_overdue_subscriptions()
    returns TABLE(gym_name character varying, expired_on date)
    language plpgsql
as
$$
BEGIN
    UPDATE gyms
    SET subscription_status = 'Suspended'
    WHERE subscription_end_date < CURRENT_DATE
      AND subscription_status = 'Active';

    RETURN QUERY
    SELECT name, subscription_end_date
    FROM gyms
    WHERE subscription_status = 'Suspended';
END;
$$;

alter function check_overdue_subscriptions() owner to root;
```

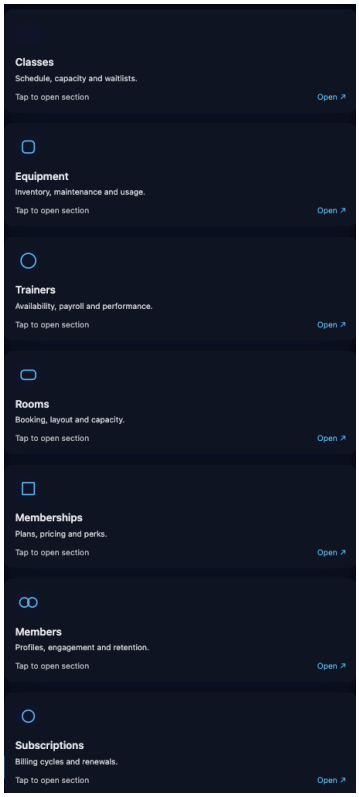
```
create function admin_archive_gym(_gym_id integer) returns text
    language plpgsql
as
$$
DECLARE
    _gym_name VARCHAR;
BEGIN
    SELECT name INTO _gym_name FROM gyms WHERE gym_id = _gym_id;

    IF _gym_name IS NULL THEN
        RETURN 'Error: Gym ID ' || _gym_id || ' not found.';
    END IF;

    UPDATE gyms
    SET subscription_status = 'Archived',
        subscription_end_date = CURRENT_DATE - 1
    WHERE gym_id = _gym_id;

    RETURN 'Success: "' || _gym_name || '" has been archived by System Admin.';
END;
$$;
```

If the gym is over due date for so long, the admin can archive the gym. Normally, after click check then the gym will be suspended immediately.

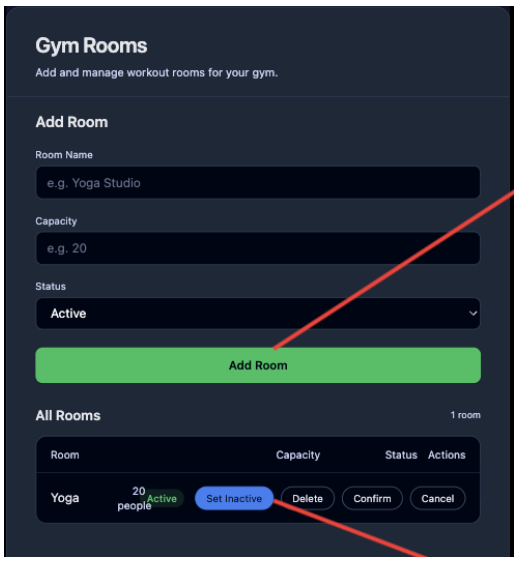
The Gym Owner : (`WHERE gym_id = _gym_id`) this helps preventing see another gym data

- Manages one or more `gyms`.

  Frontend will do this which can switch to the gym that this owner owns.



- Configures `rooms`

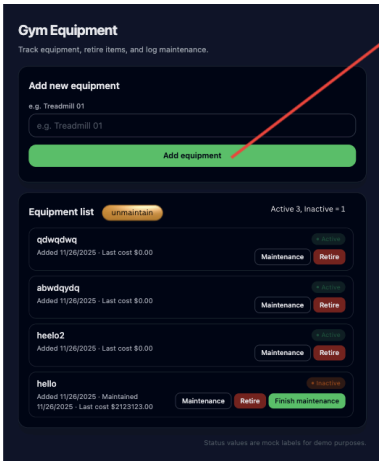



```sql
create function add_room(_gym_id integer, _name character varying, _capacity integer) returns text
    language plpgsql
as
$$
DECLARE
    _new_id INTEGER;
BEGIN
    IF _capacity <= 0 THEN
        RETURN 'Error: Capacity must be greater than 0.';
    END IF;

    IF NOT EXISTS (SELECT 1 FROM gyms WHERE gym_id = _gym_id) THEN
        RETURN 'Error: Gym ID ' || _gym_id || ' not found.';
    END IF;

    INSERT INTO rooms (gym_id, name, capacity, status)
    VALUES ( gym_id _gym_id, name _name, capacity _capacity, status 'Available')
    RETURNING room_id INTO _new_id;

    RETURN 'Success: Room "' || _name || '" added (ID: ' || _new_id || ') with capacity ' || _capacity;
```

```sql
create function get_gym_rooms(_gym_id integer)
    returns TABLE(r_room_id integer, r_name character varying, r_capacity integer, r_status character varying)
    language plpgsql
as
$$
BEGIN
    RETURN QUERY
    SELECT
        room_id,
        name,
        capacity,
        status
    FROM rooms
    WHERE gym_id = _gym_id
    ORDER BY room_id ASC;
```

- Configures `equipments`

```sql
create function perform_maintenance(_equipment_id integer, _notes text, _cost numeric) returns
    language plpgsql
as
$$
DECLARE
    _new_log_id INTEGER;
    _equip_name VARCHAR;
BEGIN
    SELECT name INTO _equip_name FROM equipment WHERE equipment_id = _equipment_id;

    IF _equip_name IS NULL THEN
        RETURN 'Error: Equipment ID ' || _equipment_id || ' not found.';
    END IF;

    INSERT INTO maintenance_logs (equipment_id, description, cost, log_date)
    VALUES (equipment_id _equipment_id, description _notes, cost _cost, log_date CURRENT_DATE)
    RETURNING log_id INTO _new_log_id;

    UPDATE equipment
    SET last_maintenance_date = CURRENT_DATE,
        status = 'Inactive'
    WHERE equipment_id = _equipment_id;

    RETURN 'Maintenance recorded. ' || _equip_name || ' is now marked as INACTIVE.';
```




```sql
create function finish_maintenance(_equipment_id integer) returns text
    language plpgsql
as
$$
DECLARE
    _equip_name VARCHAR;
BEGIN
    SELECT name INTO _equip_name FROM equipment WHERE equipment_id = _equipment_id;

    IF _equip_name IS NULL THEN
        RETURN 'Error: Equipment ID ' || _equipment_id || ' not found.';
    END IF;

    UPDATE equipment
    SET status = 'Active'
    WHERE equipment_id = _equipment_id;

    RETURN 'Success: ' || _equip_name || ' is now ACTIVE and ready for use.';
```

```sql
create function add_equipment(_gym_id integer, _name character varying, _purchase_date date DEFAULT CURRENT_DATE
    language plpgsql
as
$$
DECLARE
    _new_id INTEGER;
    _date_to_use DATE;
BEGIN
    IF NOT EXISTS (SELECT 1 FROM gyms WHERE gym_id = _gym_id) THEN
        RETURN 'Error: Gym ID ' || _gym_id || ' not found.';
    END IF;

    _date_to_use := COALESCE(_purchase_date, CURRENT_DATE);

    INSERT INTO equipment (gym_id, name, purchase_date, last_maintenance_date, status)
    VALUES ( gym_id _gym_id, name _name, purchase_date _date_to_use, last_maintenance_date NULL, status 'Active')
    RETURNING equipment_id INTO _new_id;

    RETURN 'Success: "' || _name || '" added to inventory (ID: ' || _new_id || ')';
END;
$$;
```

```sql
create function retire_equipment(_gym_id integer, _equipment_id integer) returns text
    language plpgsql
as
$$
DECLARE
    _name VARCHAR;
BEGIN
    SELECT name INTO _name
    FROM equipment
    WHERE equipment_id = _equipment_id AND gym_id = _gym_id;

    IF _name IS NULL THEN
        RETURN 'Error: Equipment not found.';
    END IF;

    UPDATE equipment
    SET status = 'Retired'
    WHERE equipment_id = _equipment_id;

    RETURN 'Success: ' || _name || ' is now marked as Retired.';
```

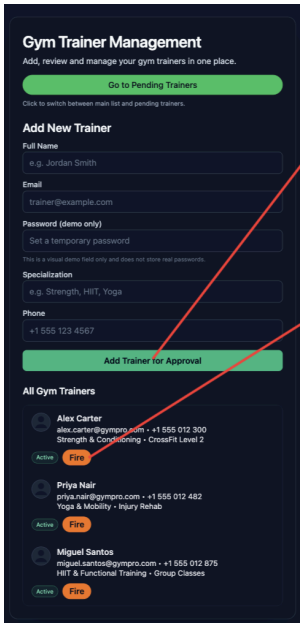
```sql
create function get_equipment_status(_gym_id integer)
    returns TABLE(r_equipment_id integer, r_name text, r_
    language plpgsql
as
$$
BEGIN
    RETURN QUERY
    SELECT
        e.equipment_id,
        e.name::TEXT,
        e.last_maintenance_date,
        e.status::TEXT
    FROM equipment AS e
    WHERE e.gym_id = _gym_id
    ORDER BY e.equipment_id;
```

```sql
create function get_equipment_summary(_gym_id integer)
    returns TABLE(status_type text, count bigint)
    language plpgsql
as
$$
BEGIN
    RETURN QUERY
    SELECT
        e.status::TEXT,
        COUNT(*)
    FROM equipment e
    WHERE e.gym_id = _gym_id
    GROUP BY e.status;
END;
```

```sql
create function get_equipment_due_for_maintenance(_gym_id integer)
    returns TABLE(r_equipment_id integer, r_name character varying, r_las
    language plpgsql
as
$$
BEGIN
    RETURN QUERY
    SELECT
        e.equipment_id,
        e.name,
        e.last_maintenance_date
    FROM equipment AS e
    WHERE e.gym_id = _gym_id
      AND (
          e.last_maintenance_date < (CURRENT_DATE - INTERVAL '30 days')
          OR
          e.last_maintenance_date IS NULL
      )
    ORDER BY e.last_maintenance_date ASC NULLS FIRST;
```

The equipment page has a lot of functions. The owner can use every function but members of the gym can only see and use `get_equipment_status` `get_equipment_summary`.

- Configure `trainers` (The gym trainer here is an approval system that after the trainer registers, the gym owner needs to review and approve the trainer. In this page members and trainers will only see the list of trainers but can not edit.)




```
create function add_trainer(_gym_id integer, _full_name character varying, _email chara
    language plpgsql
as
$$
DECLARE
    _new_id INTEGER;
BEGIN
    IF NOT EXISTS (SELECT 1 FROM gyms WHERE gym_id = _gym_id) THEN
        RETURN 'Error: Gym ID ' || _gym_id || ' not found.';
    END IF;

    IF EXISTS (SELECT 1 FROM trainers WHERE gym_id = _gym_id AND email = _email) THEN
        RETURN 'Error: Trainer email ' || _email || ' already exists in this gym.';
    END IF;

    INSERT INTO trainers (gym_id, full_name, email, password_hash, specialization, phon
    VALUES ( gym_id _gym_id, full_name _full_name, email _email, password_hash _password_
    RETURNING trainer_id INTO _new_id;

    RETURN 'Success: Trainer "' || _full_name || '" added (ID: ' || _new_id || ')';
```

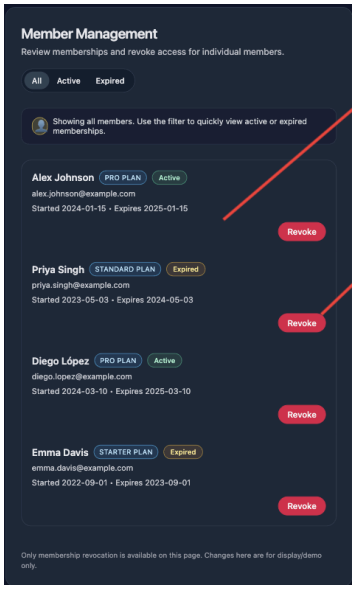
```
create function terminate_trainer(_gym_id integer, _trainer_id integer
    language plpgsql
as
$$
DECLARE
    _name VARCHAR;
BEGIN
    SELECT full_name INTO _name
    FROM trainers
    WHERE trainer_id = _trainer_id AND gym_id = _gym_id;

    IF _name IS NULL THEN
        RETURN 'Error: Trainer not found.';
    END IF;

    UPDATE trainers
    SET status = 'Terminated'
    WHERE trainer_id = _trainer_id;

    RETURN 'Success: Trainer ' || _name || ' is now removed.';
```

```
create function get_pending_trainers(_owner_id integer)
    returns TABLE(gym_name character varying, trainer_id
    language plpgsql
as
$$
BEGIN
    RETURN QUERY
    SELECT
        g.name,
        t.trainer_id,
        t.full_name,
        t.specialization,
        CURRENT_TIMESTAMP::TIMESTAMP
    FROM trainers t
    JOIN gyms g 1..n<->1: ON t.gym_id = g.gym_id
    WHERE g.owner_id = _owner_id
      AND t.status = 'Pending';
```

Once the trainer registers their status is set to pending so it shows on the pending page. After approval, the trainer now can be logging in.

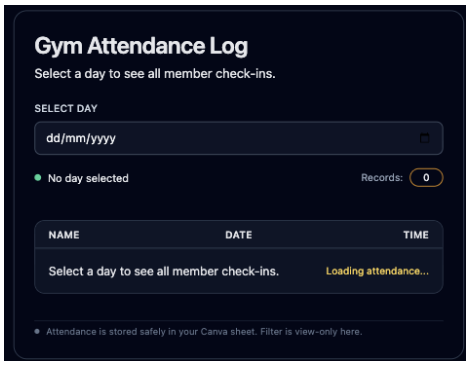
```
create function approve_trainer(_owner_id integer, _trainer_id integer) returns text
DECLARE
    _trainer_name VARCHAR;
    _gym_owner_id INTEGER;
BEGIN
    SELECT t.full_name, g.owner_id
    INTO _trainer_name, _gym_owner_id
    FROM trainers t
    JOIN gyms g 1..n<->1: ON t.gym_id = g.gym_id
    WHERE t.trainer_id = _trainer_id;

    IF _trainer_name IS NULL THEN
        RETURN 'Error: Trainer ID not found.';
    END IF;

    IF _gym_owner_id != _owner_id THEN
        RETURN 'Error: You do not have permission to approve staff for this gym.';
    END IF;

    UPDATE trainers
    SET status = 'Active'
    WHERE trainer_id = _trainer_id;

    RETURN 'Success: Trainer ' || _trainer_name || ' is now ACTIVE and can log in.';
```

- Configures `members` and `attendance log`




```
create function get_all_expirations(_gym_id integer)
 returns TABLE(r_member_id integer, r_name character varying, r_expi
    language plpgsql
as
$$
BEGIN
    RETURN QUERY
    SELECT
        m.member_id,
        m.full_name,
        m.membership_end_date,
        CASE
            WHEN m.membership_end_date < CURRENT_DATE THEN 'EXPIRED'
            ELSE 'ACTIVE'
        END::TEXT
    FROM members m
    WHERE m.gym_id = _gym_id
      AND m.membership_end_date IS NOT NULL
    ORDER BY m.membership_end_date ASC;
```

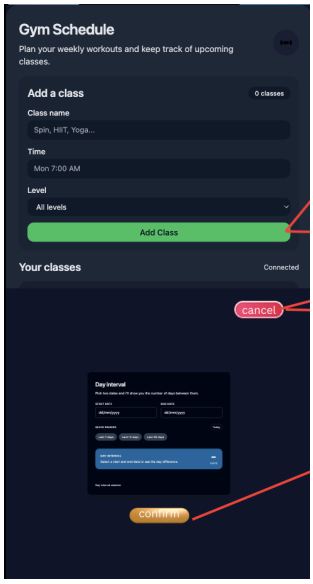
```
create function revoke_membership(_gym_id integer, _member_id integer, _reason text) re
 language plpgsql
as
$$
DECLARE
    _name VARCHAR;
BEGIN
    SELECT full_name INTO _name
    FROM members
    WHERE member_id = _member_id AND gym_id = _gym_id;

    IF _name IS NULL THEN
        RETURN 'Error: Member not found.';
    END IF;

    UPDATE members
    SET membership_end_date = CURRENT_DATE - 1
    WHERE member_id = _member_id;

    RETURN 'Success: Membership for ' || _name || ' revoked. Reason: ' || _reason;
```




```sql
create function get_daily_attendance_log(_gym_id integer, _date date DEFAULT CURRENT_DATE)
returns TABLE(r_member_name character varying, r_check_in_time timestamp without time z
language plpgsql
as
$$
BEGIN
    RETURN QUERY
    SELECT
        m.full_name,
        al.check_in_time,
        CASE
            WHEN m.membership_end_date < CURRENT_DATE THEN 'Expired'
            ELSE 'Active'
        END::VARCHAR
    FROM attendance_logs al
    JOIN members m 1..n<->1: ON al.member_id = m.member_id
    WHERE al.gym_id = _gym_id
      AND al.check_in_time::DATE = _date
    ORDER BY al.check_in_time DESC;
```

After members check in then they will be in this list and this log shows daily so it wont get all of the list.

- Configures `class`



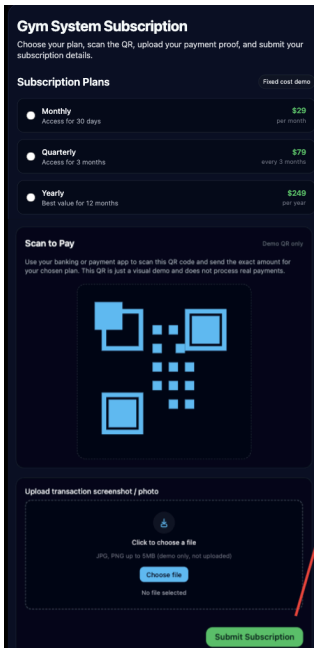

```sql
create function schedule_class(_gym_id integer, _room_id integer, _trainer_id integer, _title chara
    IF _room_status IS NULL THEN
        RETURN 'Error: Room ID ' || _room_id || ' not found.';
    END IF;

    IF _room_status != 'Available' THEN
        RETURN 'Error: Room is currently set to ' || _room_status || ' and cannot be booked.';
    END IF;

    IF EXISTS (
        SELECT 1 FROM classes
        WHERE room_id = _room_id
          AND (
                (start_time <= _start_time AND end_time > _start_time) OR
                (start_time < _end_time AND end_time >= _end_time) OR
                (start_time >= _start_time AND end_time <= _end_time)
          )
    ) THEN
        RETURN 'Error: This room is already booked during that time slot.';
    END IF;

    INSERT INTO classes (gym_id, room_id, trainer_id, title, start_time, end_time)
    VALUES ( gym_id _gym_id, room_id _room_id, trainer_id _trainer_id, title _title, start_time _start_t

    RETURN 'Success: "' || _title || '" scheduled from ' || _start_time || ' to ' || _end_time;
```

The scheduling system here prevents the time collapse.

```
create function cancel_class(_gym_id integer, _class_id integer) returns text
    language plpgsql
as
$$
DECLARE
    _title VARCHAR;
    _start_time TIMESTAMP;
BEGIN
    SELECT title, start_time INTO _title, _start_time
    FROM classes
    WHERE class_id = _class_id AND gym_id = _gym_id;

    IF _title IS NULL THEN
        RETURN 'Error: Class not found.';
    END IF;

    IF _start_time < CURRENT_TIMESTAMP THEN
        RETURN 'Error: Cannot cancel a class that has already started/finished
    END IF;

    DELETE FROM classes WHERE class_id = _class_id;
```

```
create function get_gym_schedule(_gym_id integer, _start_date
    returns TABLE(r_class_id integer, r_title character varyi
    language plpgsql
as
$$
BEGIN
    RETURN QUERY
    SELECT
        c.class_id,
        c.title,
        c.start_time,
        c.end_time,
        t.full_name AS trainer_name,
        r.name AS room_name,
        c.max_capacity
    FROM classes c
    JOIN trainers t  1..n<->1: ON c.trainer_id = t.trainer_id
    JOIN rooms r  1..n<->1: ON c.room_id = r.room_id
    WHERE c.gym_id = _gym_id
      AND c.start_time >= _start_date
      AND c.start_time <= _end_date
    ORDER BY c.start_time ASC;
```

- Configures `system_subscription_payment`

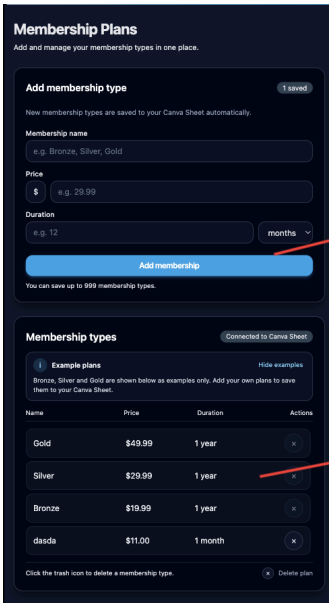



```
create function submit_payment_proof(_gym_id integer, _plan_type_id integer, _proof_ref character varying) retu
BEGIN
    SELECT name INTO _gym_name FROM gyms WHERE gym_id = _gym_id;

    IF _gym_name IS NULL THEN
        RETURN 'Error: Gym ID ' || _gym_id || ' not found.';
    END IF;

    SELECT price, name INTO _official_price, _plan_name
    FROM system_subscription_types
    WHERE type_id = _plan_type_id;

    IF _official_price IS NULL THEN
        RETURN 'Error: System Plan ID ' || _plan_type_id || ' not found.';
    END IF;

    INSERT INTO system_payment_requests (
        gym_id,
        type_id,
        transfer_amount,
        proof_reference
    )
    VALUES ( gym_id _gym_id, type_id _plan_type_id, transfer_amount _official_price, proof_reference _proof_ref);

    RETURN 'Success: Payment proof submitted for ' || _plan_name || ' ($' || _official_price || '). Please wait
```

The gym owner needs to pay and submit proof to the Admin.

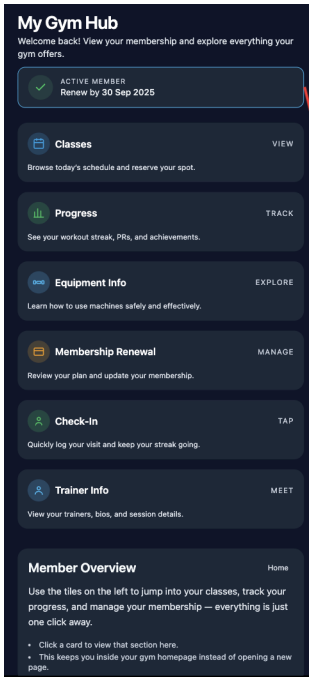
- Defines custom `membership_types` (Pricing specific to their gym).

```
create function get_gym_membership_types(_gym_id integer)
 returns TABLE(r_type_id integer, r_name character vary
    language plpgsql
as
$$
BEGIN
    RETURN QUERY
    SELECT
        type_id,
        name,
        price,
        duration_days
    FROM membership_types
    WHERE gym_id = _gym_id
    ORDER BY price ASC;
```

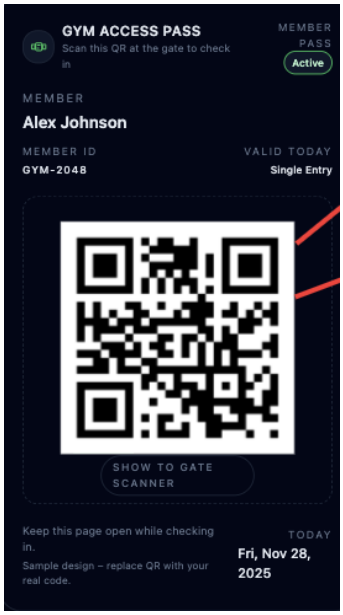



```
create function add_membership_type(_gym_id integer, _name character varying, _pric
    language plpgsql
as
$$
DECLARE
    _new_id INTEGER;
BEGIN
    IF _price < 0 THEN
        RETURN 'Error: Price cannot be negative.';
    END IF;

    IF _duration_days <= 0 THEN
        RETURN 'Error: Duration must be at least 1 day.';
    END IF;

    IF NOT EXISTS (SELECT 1 FROM gyms WHERE gym_id = _gym_id) THEN
        RETURN 'Error: Gym ID ' || _gym_id || ' not found.';
    END IF;

    INSERT INTO membership_types (gym_id, name, price, duration_days)
    VALUES ( gym_id _gym_id, name _name, price _price, duration_days _duration_days)
    RETURNING type_id INTO _new_id;

    RETURN 'Success: "' || _name || '" created (ID: ' || _new_id || ')';
```

Member Layer (The End User) :



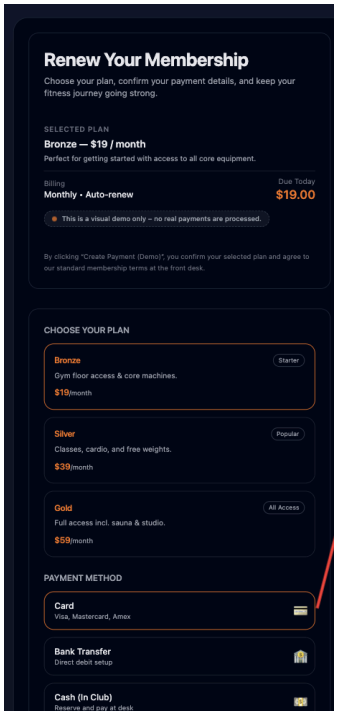
- Check in system (This add to attendance log)




```
create function check_in_member(_member_id integer, _gym_id integer) return
    _member_name VARCHAR;
BEGIN
    SELECT full_name, membership_end_date
    INTO _member_name, _end_date
    FROM members
    WHERE member_id = _member_id AND gym_id = _gym_id;

    IF _member_name IS NULL THEN
        RETURN 'Error: Member not found in this gym.';
    END IF;

    IF _end_date < CURRENT_DATE THEN
        RETURN 'ACCESS DENIED: Membership expired on ' || _end_date;
    END IF;

    INSERT INTO attendance_logs (member_id, gym_id, check_in_time)
    VALUES ( member_id _member_id, gym_id _gym_id, check_in_time CURRENT_TIME

    UPDATE members
    SET last_attend_date = CURRENT_TIMESTAMP
    WHERE member_id = _member_id;

    RETURN 'Success: Welcome ' || _member_name || '! Check-in recorded.';
```

User need to use this at gym gate

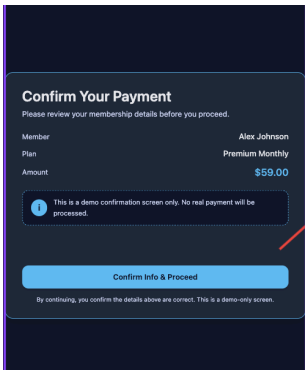
- Handles `membership renewal` (from Members to Gym Owners).




```
create function create_member_payment(_gym_id intege
        RETURN 'Error: This plan does not belong to
    END IF;

    INSERT INTO payments (
        member_id,
        type_id,
        amount,
        payment_method,
        gateway_transaction_id,
        payment_status
    )
    VALUES (
        member_id _member_id,
        type_id _plan_type_id,
        amount _official_price,
        payment_method _payment_method,
        gateway_transaction_id _transaction_id,
        payment_status 'Pending'
    )
    RETURNING payment_id INTO _new_payment_id;
```

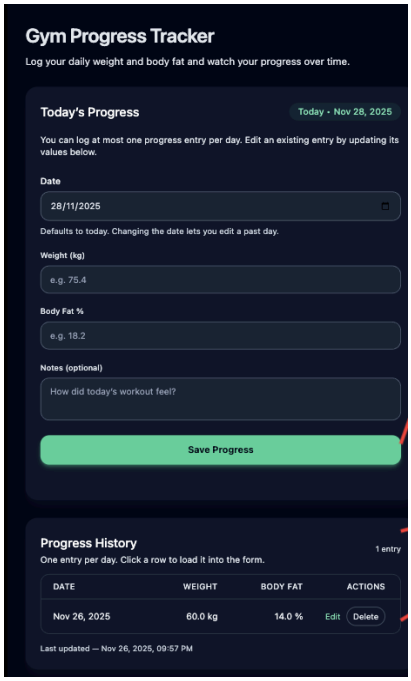
After the user clicks continue on this page and confirms payment info then the backend system will check if the transaction is complete and sure that everything is right then it will automatically renew or extend the membership.

```
create function renew_membership(_member_id integer) returns text

    IF _type_id IS NULL THEN
        RETURN 'Error: Member ID ' || _member_id || ' not found.';
    END IF;

    SELECT duration_days INTO _duration
    FROM membership_types
    WHERE type_id = _type_id;

    IF _duration IS NULL THEN
        RETURN 'Error: Membership Type not configured properly.';
    END IF;

    IF _current_end IS NULL OR _current_end < CURRENT_DATE THEN
        _new_end := CURRENT_DATE + _duration;
    ELSE
        _new_end := _current_end + _duration;
    END IF;

    UPDATE members
    SET membership_end_date = _new_end
    WHERE member_id = _member_id;

    RETURN 'Success: Membership renewed. New Expiry: ' || _new_end;
```




```
INTO _member_id, _amount, _current_status
FROM payments
WHERE payment_id = _payment_id;

IF _member_id IS NULL THEN
    RETURN 'Error: Payment ID ' || _payment_id || ' not found.';
END IF;

IF _current_status = 'Paid' THEN
    RETURN 'Error: This payment has already been processed.';
END IF;

IF _current_status != 'Pending' THEN
    RETURN 'Error: Payment status is ' || _current_status || '. Cannot process.';
END IF;

UPDATE payments
SET payment_status = 'Paid'
WHERE payment_id = _payment_id;

PERFORM renew_membership(_member_id _member_id);

RETURN 'Success: Payment verified ($' || _amount || ') and membership extended.';
```

- Class info : The member will only see the class list and can not edit it.
- Equipment info : The member will only see the equipment list and its status and can not edit it.
- Log `progress`

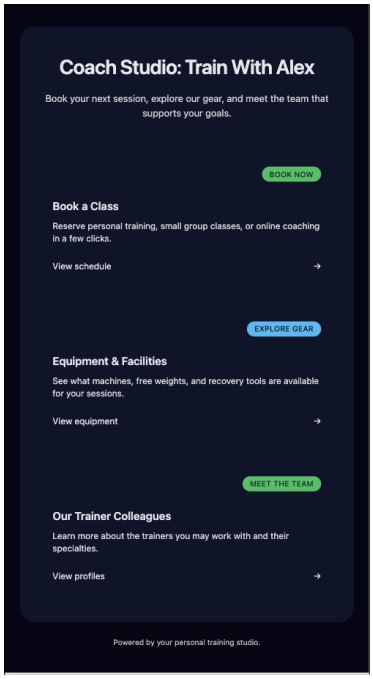



```sql
create function log_progress(_member_id integer, _weight numeric, _body_fat nume
    language plpgsql
as
$$
DECLARE
    _member_name VARCHAR;
BEGIN
    SELECT full_name INTO _member_name FROM members WHERE member_id = _member_id

    IF _member_name IS NULL THEN
        RETURN 'Error: Member ID ' || _member_id || ' does not exist.';
    END IF;

    INSERT INTO progress_logs (member_id, log_date, weight_kg, body_fat_percent)
    VALUES ( member_id _member_id, log_date CURRENT_DATE, weight_kg _weight, bod
    ON CONFLICT (member_id, log_date)
    DO UPDATE SET
        weight_kg = EXCLUDED.weight_kg,
        body_fat_percent = EXCLUDED.body_fat_percent;

    RETURN 'Success: Progress for ' || _member_name || ' updated for today.';
END;
```

```sql
create function get_member_progress(_gym_id integer, _member_id integer)
    returns TABLE(log_date date, weight_kg numeric, body_fat_percent numeric)
    language plpgsql
as
$$
BEGIN
    RETURN QUERY
    SELECT
        pl.log_date,
        pl.weight_kg,
        pl.body_fat_percent
    FROM progress_logs pl
    JOIN members m  1..n<->1: ON pl.member_id = m.member_id
    WHERE pl.member_id = _member_id
      AND m.gym_id = _gym_id
    ORDER BY pl.log_date ASC;
END;
```

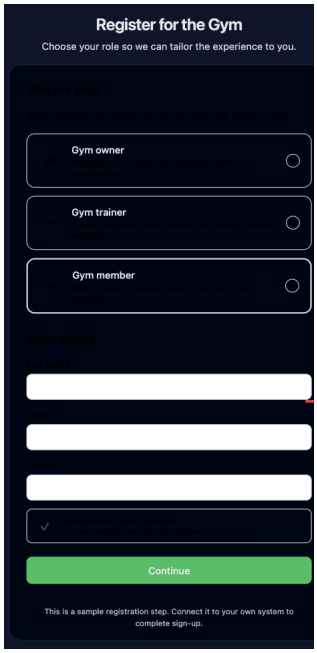
In one day there must be only 1 log so if the member tries to log in that day it will update the log progress. (I think that one day the human body won't have a lot of change.)

Staff Layer (Trainers) :



- configures `class` : The trainer will have the same permission as the gym owner to schedule class and cancel the class.
- `Equipment info, Trainer page` : Trainer also sees equipment and trainer information like members of the gym and they can not edit.

Register and Login : (The password security, I assume that in backend it already has the encrypt method so it assures that the data travels between app,web and database are securely safe.)



They need to choose what position they want then type all required information.

Unauthorized access to sensitive member data, a **Trainer Approval Workflow :**

1. Trainers register via a self-service portal (Status: `Pending`). They can login but cant access their dashboard

```
create function register_trainer(_gym_id integer, _full_name character varying, _email character
    language plpgsql
as
$$
DECLARE
    _gym_name VARCHAR;
BEGIN
    SELECT name INTO _gym_name FROM gyms WHERE gym_id = _gym_id;

    IF _gym_name IS NULL THEN
        RETURN 'Error: Gym ID ' || _gym_id || ' not found.';
    END IF;

    IF EXISTS (SELECT 1 FROM trainers WHERE gym_id = _gym_id AND email = _email) THEN
        RETURN 'Error: Email already registered at this gym.';
    END IF;

    INSERT INTO trainers (gym_id, full_name, email, password_hash, specialization, phone, status)
    VALUES ( gym_id _gym_id, full_name _full_name, email _email, password_hash _password_hash, spe

    RETURN 'Success: Application sent to ' || _gym_name || '. Please wait for Owner approval.';
```

```
create function login_trainer(_gym_id integer, _email character varying, _password_hash character varying)
    _id INT;
    _name VARCHAR;
BEGIN
    SELECT trainer_id, full_name, status
    INTO _id, _name, _status
    FROM trainers
    WHERE gym_id = _gym_id
      AND email = _email
      AND password_hash = _password_hash;

    IF _id IS NULL THEN
        RETURN QUERY SELECT NULL::INT, NULL::VARCHAR, NULL::VARCHAR, 'Login Failed: Invalid credentials'::TEXT;

    ELSIF _status = 'Pending' THEN
        RETURN QUERY SELECT _id, _name, _status, 'Login Failed: Account is pending approval'::TEXT;

    ELSIF _status = 'Terminated' THEN
        RETURN QUERY SELECT _id, _name, _status, 'Login Failed: Access revoked'::TEXT;

    ELSE
        RETURN QUERY SELECT _id, _name, _status, 'Success'::TEXT;
    END IF;
```

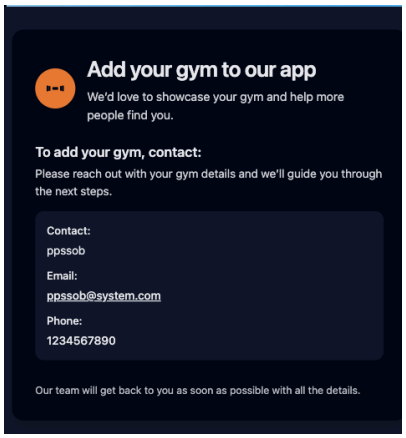
2. Gym Owners receive a dashboard notification.
3. Owners must explicitly call `approve_trainer()` to grant login access.

A **Gym Owner Application Workflow :**

1. Gym owner registers (No gym).

```
create function register_gym_owner(_full_name character varying, _email character varying, _pa
 language plpgsql
as
$$
DECLARE
    _new_id INTEGER;
BEGIN
    IF EXISTS (SELECT 1 FROM gym_owners WHERE email = _email) THEN
        RETURN 'Error: Email ' || _email || ' is already registered.';
    END IF;

    INSERT INTO gym_owners (full_name, email, password_hash, phone)
    VALUES ( full_name _full_name,  email _email,  password_hash _password_hash,  phone _phone)
    RETURNING owner_id INTO _new_id;

    RETURN 'Success: Owner registered with ID: ' || _new_id;
```

2. Login and they will see the contact information pop-up to add their gym (This will make sure that their gym has everything set up before applying this system and also about the system contract that they need to sign.)



3. Admin will explicitly do add new gym so the gym is entering the flow.

A **Gym Member Application Workflow :**

1. user register as member and login (Status: `Pending`)

```sql
create function register_member(_gym_id integer, _name character varying, _email character varying, _pass
    IF EXISTS (
        SELECT 1
        FROM members
        WHERE gym_id = _gym_id
          AND email = _email
    ) THEN
        RETURN 'Error: The email ' || _email || ' is already registered in this gym.';
    END IF;

    INSERT INTO members (
        gym_id,
        full_name,
        email,
        password_hash,
        phone,
        type_id,
        membership_end_date
    )
    VALUES (
        gym_id _gym_id,
        full_name _name,
        email _email,
        password_hash _password_hash,
        phone _phone,
        type_id _type_id,
        membership_end_date NULL
    );

    RETURN 'Success: Account created for ' || _name || '. Please make a payment to activate membership.';
```

2. After login it should pop up the payment page for selecting plan type and payment method. After confirming and everything is right then now their membership starts.