CS2102 Database Systems

Project Team 4

Pet Caring Service (PCS)

James Lee (A0200300N)
Kong Jian Wei (A0200284U)
Jason Chee (A0183952W)
Lim Zheng Wei (A0180324N)
Koh Vinleon (A0202155W)

Project Responsibilities

Development		Responsibilities	Assigned To
Database Development		Hosting of Database and Creation of Tables	Jian Wei
		Creation of Trivial Functions and Procedure	Vinleon
		Creation of Data Constraint Triggers	Vinleon Jian Wei James
Backend Development		Setting up and development of backend services	Vinleon
		Setting up of notification services	James
		Backend Calculation (Bid cost, Salary, Bonus etc.)	Jian Wei
Web Development	User Authentication	1. Login Page 2. Registration Page 3. Profile Page	Jian Wei
	Administrator Console	1. Administrator Management 2. Pet Owners Management 3. Part-timers Management 4. Full-timers Management	Jian Wei
		5. Dashboard 6. Pet Category Management	Vinleon
	Pet Owners Console	1. Pet Management 2. Bidding for caretakers 3. Rate/review	James
	Caretakers Console	1. Indicate Availability (PT) 2. Apply Leave (FT)	Vinleon
		Bids Management (Accept, View Ongoing Job)	Zheng Wei
		Profile Page (View Salary Earned, Ratings and Reviews, Edit Kind of Pets)	Jason

Data Requirements and Functionalities

Overview

The pet caring service (PCS) application is a platform that allows caretakers to offer pet sitting services for pet owners.

Core Functionalities

Common

- Users are able to login
- Users are able to sign up as a pet owner, caretaker or both
- Users are able to update their profile
- Users are able to change their password

Pet Owner

- Pet Owners are able to add / update / delete their pets
- Pet Owners are able to find caretakers for their pets
- Pet Owners are able to bid for caretaker services
- Pet Owners are able to review caretakers

Caretakers

- Caretakers can be a part-time or full-time caretaker
- Caretakers are able to state their availability/leave dates
- Caretakers are able to specify which categories of pets they can take care of
- Caretakers are able to accept bids from pet owners
- Caretakers are able to see their ratings and reviews

PCS Admin

- Admins are able to update/delete accounts
- Admins are able to to create new admin
- Admins are able to to create new pet category

Data Constraints

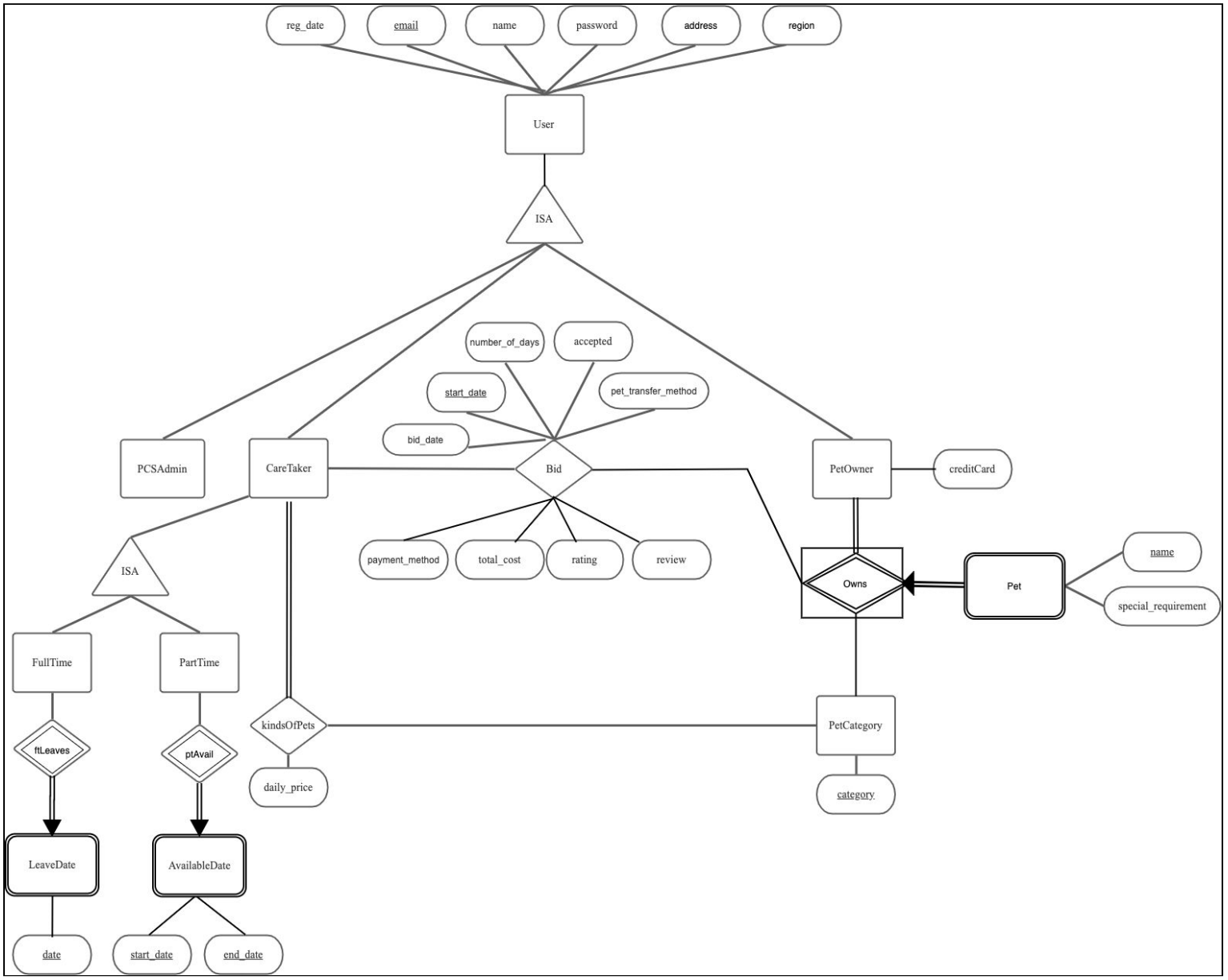
- **Users** are identified by their email. Their name, address, email, and password must also be recorded.
- Every user must be a **Pet Owner** or **Caretaker**, or both, or **PCS Admin**. (If you are PCS admin, you cannot be any other type).
- **Pet Owners** must have at least one Pet and recorded preferred payment method. Their credit card details may also be stored.
- **Pets** are identified by their **pet owner's email and pet's name**. Their name and category must also be recorded. Their **special requirements** may also be recorded.
- **Pet Owners** can bid for a particular **Caretaker** on available dates for a certain number of days. Preferred pet transfer method must also be recorded.
- **Caretakers** can choose to accept a bid
- Every **bid** will be identified by the Pet Owner's email, pet's name, start date and Caretaker's email. The total cost, date time and the payment method must be recorded.
- **Pet Owners** can post ratings and reviews for a specific **bid**.
- Each **Caretaker** must be either **full-time** or **part-time**.
- **Caretakers** must have a daily price for every **pet**.
- **Caretakers** cannot take care of their own **pets** (if they are also a **Pet Owner**).
- **Caretakers** should only take care of **pets** that they can care for.
- **Caretakers** should not have overlap between their availability / leave periods.
- **Part-time Caretakers** can specify their availability for current and next year.

- **Full-time Caretakers** must be available to work for a minimum of 2 times 150 consecutive days per year.
- **Full-time Caretakers** can take leave except when they have one or more pets under their care.
- A **Full-time Caretaker** must not take care of more than 5 pets in one period.
- A **Part-time Caretaker** must not take care of more than 2 pets in one period, unless they have a good rating, then they can have up to 5.

Interesting / Non-Trivial Aspects

- **ISA User Relationship** is used between **PCS Admin**, **Caretakers** and **Pet Owners** to enforce covering, overlap between Caretakers and Pet Owners and non-overlap between PCS Admin and the others.
- **ISA Caretaker Relationship** is used between **Full-Time** and **Part-Time** implemented to enforce non-overlap relationships.
- **Pet Owner**'s ownership of pets is implemented as an **Aggregation**.
- **Pet** is implemented as **Identity Dependency Weak Entity**, as pets are unable to represent themselves without the **Pet Owners**.
- **Full-Timer**'s leaves are implemented as **Identity Dependency Weak Entity**, as leaves are unable to represent themselves without a **Full-Timer**.
- **Part-Timer**'s availability dates are implemented as Identity Dependency Weak Entity, as availability dates are unable to represent themselves without a **Part-Timer**.
- **Users'** passwords are encrypted.
- **Pet Owners** can search for a caretaker to take care of all of his/her pets for a certain time period.
- **Pet Owners** get notified when bids are accepted.
- **Pet Owners** can bid for multiple caretakers for the same pet with overlapping time periods, but only one will be accepted.
- **Caretakers** can receive multiple bids that have overlapping time periods, but will only be able to accept a maximum of **5** (Full-Timers and Part-Timers with average rating of 4) or **2** (Regular Part-Timer).
- **Caretakers** are able to look at their salary report including their bonuses at any time.
- When a **Caretaker** is no longer with the company but has future jobs, a replacement Full-Timer Caretaker will take over the job.
- **Pet Owners** and **PCS Admin** can view the top performing **Caretakers**.
- **PCS Admin** is able to view the performance of the company (new registrations, new bids and revenue).

ER Model



Constraints Not Captured on ER Diagram

1. Overlapping/covering constraints not captured.
   a. **Users ISA** (PCS Admin / Caretaker / Pet Owner)
      i. **Overlapping** allowed between **Caretakers** and **Pet Owners**
      ii. **Overlapping** not allowed between **PCS Admin** and **other types**
      iii. **Covering** Constraint
   b. **Caretakers ISA** (Full-Timer / Part-Timer)
      i. Overlapping not allowed between **full-timer** and **part-timer**
      ii. **Covering** Constraint
2. **Caretakers** cannot take care of their own **pets** (if they have a **Pet Owner** account)
3. **Caretakers** should only take care of **pets** that they can care for.
4. **Caretakers** should not have overlap between their **availability / leave** periods
5. **Part-time Caretakers** can specify their **availability** for current and next year
6. **Full-time Caretakers** must work for a minimum of **2 times 150** consecutive days per year

7. **Full-time Caretakers** can take leave except when they have one or more pets under their care
8. A **Full-time Caretaker** must not take care of more than 5 pets in one period.
9. A **Part-time Caretaker** must not take care of more than 2 pets in one period, unless they have a good rating, then they can have up to 5.

Relational Schema

Tables	Constraints Enforced	3NF/BCNF
DROP TABLE IF EXISTS pcs_user **CASCADE**; **CREATE TABLE** pcs_user (email VARCHAR(320) **PRIMARY KEY**, name VARCHAR(50) **NOT NULL**, password VARCHAR(50) **NOT NULL**, address VARCHAR(300), reg_date TIMESTAMP **DEFAULT** now());	**Primary Key** (email) A user only can have only one account on the PCS website. **Constraints** The existence of pcs_user table allows enforcement of the covering constraint.	BCNF
DROP TABLE IF EXISTS pcs_admin **CASCADE**; **CREATE TABLE** pcs_admin (email VARCHAR(320) **PRIMARY KEY**, **FOREIGN KEY** (email) **REFERENCES** pcs_user (email) **ON UPDATE CASCADE ON DELETE CASCADE**);	**Primary Key (**email**)** Email is used to uniquely identify each admin. **Foreign References** email (references pcs_user) An admin must be registered as a user first. *(ISA child of pcs_user).*	BCNF
DROP TABLE IF EXISTS care_taker **CASCADE**; **CREATE TABLE** care_taker (email VARCHAR(320) **PRIMARY KEY**, **FOREIGN KEY** (email) **REFERENCES** pcs_user (email) **ON UPDATE CASCADE ON DELETE CASCADE**);	**Primary Key** (email) Email is used to uniquely identify each caretaker. **Foreign References** email (references pcs_user) A caretaker must be registered as a user first. (ISA child of user). **Constraints** The existence of the care_taker table allows enforcement of the covering constraint.	BCNF
DROP TABLE IF EXISTS full_timer **CASCADE**; **CREATE TABLE** full_timer (email VARCHAR(320) **PRIMARY KEY**, **FOREIGN KEY** (email) **REFERENCES** care_taker (email) **ON UPDATE CASCADE ON DELETE CASCADE**);	**Primary Key** (email) Email is used to uniquely identify each full-timer. **Foreign References** email (references care_taker) A full-timer must be a caretaker first. *(ISA child of care_taker).*	BCNF

DROP TABLE IF EXISTS part_timer **CASCADE**; **CREATE TABLE** part_timer (email VARCHAR(320) **PRIMARY KEY**, **FOREIGN KEY** (email) **REFERENCES** care_taker (email) **ON UPDATE CASCADE ON DELETE CASCADE**);	**Primary Key** (email) Email is used to uniquely identify each part-timer. **Foreign References** email (references care_taker) A part-timer must be a caretaker first. *(ISA child of care_taker).*	BCNF
DROP TABLE IF EXISTS ft_leaves **CASCADE**; **CREATE TABLE** ft_leaves (email VARCHAR(320), date DATE, **PRIMARY KEY** (email, date), **FOREIGN KEY** (email) **REFERENCES** full_timer (email) **ON UPDATE CASCADE ON DELETE CASCADE,** **CONSTRAINT** date_check **CHECK** (date > now()));	**Primary Key** (email, date) A full-timer's leave is uniquely identified by its email and leave date. **Foreign References** email (references full_timer) A full-timer must exist before leave can be created. **Constraints** A full-time caretaker cannot past date leave.	BCNF
DROP TABLE IF EXISTS pt_availability **CASCADE**; **CREATE TABLE** pt_availability (email VARCHAR(320), start_date DATE, end_date DATE, **PRIMARY KEY** (email, start_date, end_date), **FOREIGN KEY** (email) **REFERENCES** part_timer (email) **ON UPDATE CASCADE ON DELETE CASCADE,** **CONSTRAINT** end_date_check **CHECK** (end_date <= (now() + interval '2 years')), **CONSTRAINT** start_date_check **CHECK** (start_date > now()));	**Primary Key** (email, start_date, end_date) A part-timer's availability is uniquely identified by its email, start date and end date. **Foreign References** email (references part_timer) A part-timer must exist before availability can be created. **Constraints** start_date A part-timer's available dates cannot be in the past. end_date Part-timers can only declare their availability for up to 2 years from the present time.	BCNF
DROP TABLE IF EXISTS pet_owner **CASCADE**; **CREATE TABLE** pet_owner (email VARCHAR(320) **PRIMARY KEY**, credit_card VARCHAR(16), **FOREIGN KEY** (email) **REFERENCES** pcs_user (email) **ON UPDATE CASCADE ON DELETE CASCADE**);	**Primary Key** (email) Email is used to uniquely identify each pet owner. **Foreign References** email (references pcs_user) A caretaker must be registered as a user first. *(ISA child of pcs_user).* **Constraints**	BCNF

	The total participation constraint with pets is enforced through insert_pet_owner_with_pet() function.	
DROP TABLE IF EXISTS pet_category **CASCADE**; **CREATE TABLE** pet_category (category VARCHAR(50) **PRIMARY KEY**);	**Primary Key** (category) The category name is used to uniquely identify each pet category.	BCNF
DROP TABLE IF EXISTS pet_owns **CASCADE**; **CREATE TABLE** pet_owns (po_email VARCHAR(320), name VARCHAR(320), category VARCHAR(50), special_requirement VARCHAR(1000), **PRIMARY KEY** (po_email, name), **FOREIGN KEY** (po_email) **REFERENCES** pet_owner(email) **ON UPDATE CASCADE ON DELETE CASCADE**, **FOREIGN KEY** (category) **REFERENCES** pet_category(category) **ON UPDATE CASCADE**);	**Primary Key** (po_email, name) Ownership of pets is referenced by the pet owner's email and the pet's name since its name cannot uniquely identify a pet. **Foreign References** po_email (references pet_owner) A pet owner must exist before an ownership can be established. category (references pet_category) The category of the pet must exist before an ownership can be established.	BCNF
DROP TABLE IF EXISTS kind_of_pets **CASCADE**; **CREATE TABLE** kind_of_pets (ct_email VARCHAR(320), category VARCHAR(50), daily_price NUMERIC(3,2) **NOT NULL**, **PRIMARY KEY** (ct_email, category), **FOREIGN KEY** (ct_email) **REFERENCES** care_taker(email), **FOREIGN KEY** (category) **REFERENCES** pet_category(category), **CONSTRAINT** daily_price_check **CHECK** (daily_price > 0));	**Primary Key** (ct_email, category) Kind of Pets is used to identify what category of pets a caretaker can care for, identified by his/her email and the category. **Foreign References** ct_email (references care_taker), category (references pet_category) In order to create a kind of pets entry, the caretaker and pet category must exist. **Constraints** The daily_price must not be $0.	BCNF
DROP TABLE IF EXISTS bid **CASCADE**; **CREATE TABLE** bid (po_email VARCHAR(320) **NOT NULL**, pet_name VARCHAR(320) **NOT NULL**, ct_email VARCHAR(320) **NOT NULL**, start_date DATE **NOT NULL**, number_of_days INTEGER **NOT NULL**, status VARCHAR(20) **NOT NULL DEFAULT** 'Pending', pet_transfer_method VARCHAR(300) **NOT NULL**,	**Primary Key** (po_email, pet_name, ct_email, start_date) These 4 attributes are used to identify each bid in the table uniquely. With these PKs in place, more than one bid can be placed for the same caretaker by the same pet owner for the same pet, albeit on different start dates. **Foreign References** po_email, name (references pet_owner) In order to create a bid, the pet	BCNF

payment_method VARCHAR(50), total_cost NUMERIC(5,2), rating INTEGER, review VARCHAR(1000), bid_date TIMESTAMP **NOT NULL DEFAULT** now(), **PRIMARY KEY** (ct_email, pet_name, po_email, start_date), **FOREIGN KEY** (ct_email) **REFERENCES** care_taker (email), **FOREIGN KEY** (pet_name, po_email) **REFERENCES** pet_owns (name, po_email) **ON UPDATE CASCADE ON DELETE CASCADE**, **CONSTRAINT** bid_number_of_days_check **CHECK** (number_of_days > 0), **CONSTRAINT** bid_rating_check **CHECK** (rating >= 0 AND rating <= 5), **CONSTRAINT** bid_ct_po_email_check **CHECK** (ct_email <> po_email), **CONSTRAINT** bid_payment_method **CHECK** (payment_method = **ANY** (**ARRAY**['Cash', 'Credit Card'])));	owner must be registered and owns a pet. ct_email (references care_taker) In order to bid a caretaker, the caretaker must be registered as well. **Constraints** number_of_days to engage pet caring service have to be minimum 1. rating should be between 0 to 5 ct_email and po_email must not be the same, as a pet owner cannot take care of it's own pet. payment_method should be either cash or credit card.	

Constraints Enforced By Triggers

Table Name	Constraints	Trigger Created
pcs_admin	A **Caretaker** or **Pet Owner** cannot be a **PCS Admin** at the same time. (No Overlap Constraint)	**BEFORE INSERT** check_admin_role()
care_taker	When a **Caretaker** is removed from the table, check if he/she is a **Pet Owner**. If he/she is not, remove it from the **PCS User** table. (Covering Constraint)	**AFTER DELETE** check_ct_secondary_role()
	A **PCS Admin** cannot be a **Caretaker** at the same time. (No Overlap Constraint)	**BEFORE INSERT** check_ct_role()
pet_owner	When **Pet Owner** is removed from the table, check if he/she is a **Caretaker**. If he/she is not, remove it from the **PCS User** table. (Covering Constraint)	**AFTER DELETE** check_po_secondary_role()
	A **PCS Admin** cannot be a **Pet Owner** at the same time. (No Overlap Constraint)	**BEFORE INSERT** check_po_role()
full_timer	A **Caretaker** cannot be **full-time** and **part-timer** at the same time. (No Overlap Constraint)	**BEFORE INSERT** check_ft_ct_role()
part_timer	A **Caretaker** cannot be **full-time** and **part-time** at the same time. (No Overlap Constraint)	**BEFORE INSERT** check_pt_ct_role()

ft_leaves	**A full-time Caretaker** can take leave except when they have one or more pets under their care. **Full-time Caretakers** must be available to work for a minimum of 2 times 150 consecutive days per year.	**BEFORE INSERT** check_valid_leave_date()
pt_availability	**Part-time Caretakers** should not have overlap between their availability periods.	**BEFORE INSERT** check_valid_avail_date()
bids	A **full-time Caretaker** must not take care of more than 5 pets in one period. A **part-time Caretaker** must not take care of more than 2 pets in one period, unless they have a good rating, then they can have up to 5.	**BEFORE UPDATE** check_ct_pet_count()
	A **Pet Owner** is allowed to bid for a **Caretaker** when the **Caretaker** has more than the maximum pet bids in "pending" status. However, **Pet Owners** are not allowed to bid for **Caretaker** that has accepted the maximum number (2 or 5) of bids on the specific date.	**BEFORE INSERT** check_ct_pet_count_before_insert()
	Caretakers should only take care of **pets** that they can care for.	**BEFORE INSERT** check_ct_suitability()
	Pet Owners are not allowed to create bids for today or any dates earlier.	**BEFORE INSERT** check_bid_date()
	Automated updation on **Bid** Table on the total cost of inserted bid.	**AFTER INSERT** calculate_bid_cost()
	A **full-time Caretaker** will always accept the job immediately if possible	**BEFORE INSERT** check_bid_acceptance()
kind_of_pets	**Caretakers** should only take care of **pets** that they can care for. If they choose to stop taking care of a pet category, unconfirmed bids will be rejected.	**BEFORE DELETE** check_on_kind_of_pets_delete()

Non-Trivial/Interesting Triggers

Some of the triggers and complex queries use the following functions to aid in the computation of end date and the retrieval of Caretaker's average ratings respectively.

```
DROP FUNCTION IF EXISTS get_end_date(DATE, INTEGER);
CREATE OR REPLACE FUNCTION get_end_date(startdate DATE, numberofdays INTEGER)
   RETURNS DATE
AS $$
DECLARE end_date DATE;
BEGIN
SELECT (startdate + interval '1 day' * (numberofdays-1)) INTO end_date;
        RETURN end_date;
END; $$
LANGUAGE 'plpgsql';
```

```
DROP FUNCTION IF EXISTS get_avg_ratings(VARCHAR);
CREATE OR REPLACE FUNCTION get_avg_ratings(ctemail VARCHAR)
   RETURNS NUMERIC
AS $$
DECLARE computed_rating NUMERIC;
BEGIN
        SELECT ROUND(COALESCE ((SELECT AVG(rating) FROM Bid WHERE ct_email = ctemail), 0), 2) INTO
        computed_rating;
        RETURN computed_rating;
END; $$
LANGUAGE 'plpgsql';
```

1. **Reassigning of Caretaker**

This trigger is written for the situation when a Caretaker is deleted from the database *(for various reasoning, e.g. terminated, resigned, unforeseen circumstances, etc.)*, there must be a replacement full-time Caretaker to take over his/her upcoming bids (since only full-timers are able to auto-accept bids according to project requirement).

It will retrieve all the upcoming bids (bids where the start date is after the date of deletion) of the Caretaker that is to be deleted. Then, it will perform a loop through all his/her bids and search for the next most suitable Caretaker. The replacement Caretaker, which referred to as a candidate, must meet the following criteria:

1. Must be a full-time Caretaker (since only full-timer can auto-accept jobs)
2. Must not be on leave during the requested dates
3. Must be under Caretaker's selected pet category
4. Must not take care of more than 5 pets during the requested dates

If there's more than one suitable candidate, sort by nearest region, then sort by ratings.
If such a candidate exists, the function will update the bid's record. Else, the status of the bid will be changed to "Rejected" as an indicator to the Pet Owner that the bid is no longer valid.

The trigger is activated before deletion of Caretaker, primarily to update the Bid record before we delete to ensure the data is not lost.

```
DROP FUNCTION IF EXISTS reassigning_care_taker();
```

```
CREATE OR REPLACE FUNCTION reassigning_care_taker()
RETURNS TRIGGER
AS $$
DECLARE old_bid_row bid;
DECLARE selected_ct VARCHAR;

BEGIN
-- Retrieving the deleted Caretaker's future bids that are accepted
FOR old_bid_row IN SELECT * FROM bid B WHERE B.ct_email = OLD.email
AND B.status = 'Accepted' AND B.start_date > now()
LOOP
        SELECT FT.email FROM full_timer FT NATURAL JOIN care_taker NATURAL JOIN
        pcs_user PU
        -- Must not be on leave during the requested dates
        WHERE FT.email NOT IN (SELECT FL.email FROM ft_leaves FL WHERE FL.date >=
        old_bid_row.start_date AND FL.date <= (SELECT get_end_date(old_bid_row.start_date,
        old_bid_row.number_of_days)))
        -- Check against Caretaker's Pet Categories
        AND (SELECT category FROM pet_owns PO WHERE PO.po_email = old_bid_row.po_email
        AND PO.name = old_bid_row.pet_name) IN
        (SELECT category FROM kind_of_pets KP WHERE KP.ct_email = FT.email)
        -- Check the Pet Limit
        AND (SELECT COUNT(*) FROM bid B WHERE B.ct_email = FT.email
        AND (old_bid_row.start_date BETWEEN B.start_date
        AND (SELECT get_end_date(B.start_date, B.number_of_days))
        OR B.start_date BETWEEN old_bid_row.start_date
        AND (SELECT get_end_date(old_bid_row.start_date, old_bid_row.number_of_days)))) < 5
        -- Sort by region descendingly, then ratings descendingly
        ORDER BY (CASE WHEN PU.region = (SELECT region FROM pet_owner PO NATURAL JOIN
        pcs_user WHERE PO.email = old_bid_row.po_email) THEN 1 ELSE 0 END) DESC, (SELECT
        get_avg_ratings(FT.email)) DESC
        LIMIT 1
        INTO selected_ct;
        IF selected_ct IS NOT NULL THEN
                UPDATE bid SET ct_email = selected_ct
                AND total_cost = (SELECT daily_price FROM kind_of_pets KP WHERE KP.ct_email =
                selected_ct AND KP.category = (SELECT category FROM pet_owns PO WHERE
                PO.po_email = old_bid_row.po_email AND PO.name = old_bid_row.pet_name)) *
                old_bid_row.number_of_days
                WHERE ct_email = old_bid_row.ct_email
                AND start_date = old_bid_row.start_date
                AND po_email = old_bid_row.po_email
                AND pet_name = old_bid_row.pet_name;
        ELSE
                UPDATE bid SET status = 'Rejected'
                WHERE ct_email = old_bid_row.ct_email
                AND start_date = old_bid_row.start_date
                AND po_email = old_bid_row.po_email
```

```
                AND pet_name = old_bid_row.pet_name;
        END IF;
        END LOOP;
    RETURN NEW;
END; $$
LANGUAGE 'plpgsql';

DROP TRIGGER IF EXISTS delete_ct_trigger ON care_taker;
CREATE TRIGGER delete_ct_trigger
    BEFORE DELETE
    ON care_taker
    FOR EACH ROW
    EXECUTE PROCEDURE reassigning_care_taker();
```

2. **Auto Rejection of Overlapping Bids upon Acceptance**

This trigger was written for auto rejecting bids when:

1. The Caretakers when they reached the maximum number of pets they can take care of.
2. Pets that have overlap dates bids that are pending with other Caretakers.

The trigger is activated after a bid update. Upon the bid acceptance, the trigger function will check how many pets is the caretaker taking care of that overlaps with the newly accepted bid. If he/she exceeds the number of pets that they can take care of (5 for Full-Timer, 5 for Part-Timer with good average ratings of 4, and 2 for regular Part-Timer), we will retrieve all the pending overlapped bids and set the status to “Rejected”.

Afterwards, it will check and retrieve the pet's bids that have overlap dates bids that are pending with other Caretakers and set the status to "Rejected" as well.

```
DROP FUNCTION IF EXISTS check_on_bid_accept();
CREATE FUNCTION check_on_bid_accept()
RETURNS TRIGGER
AS $$
DECLARE bid_count INTEGER;
DECLARE bid_row bid;
BEGIN
        SELECT COUNT(*) FROM bid B WHERE B.ct_email = NEW.ct_email AND B.status = 'Accepted'
        AND (NEW.start_date BETWEEN B.start_date AND (SELECT get_end_date(B.start_date,
        B.number_of_days))
        OR B.start_date BETWEEN NEW.start_date AND (SELECT get_end_date(NEW.start_date,
        NEW.number_of_days))) INTO bid_count;

        IF bid_count = (CASE WHEN (SELECT COUNT(*) FROM full_timer FT WHERE FT.email =
                        NEW.ct_email) > 0 THEN 5
                                WHEN (SELECT COUNT(*) FROM part_timer PT WHERE PT.email =
                        NEW.ct_email) > 0 THEN
                                (CASE WHEN (SELECT get_avg_ratings (NEW.ct_email)) >= 4
                                        THEN 5 ELSE 2 END) END)
```

```
        THEN
        FOR bid_row IN SELECT * FROM bid B1 WHERE B1.status = 'Pending'
        AND B1.ct_email = NEW.ct_email AND (B1.start_date BETWEEN NEW.start_date AND
        (SELECT get_end_date(NEW.start_date, NEW.number_of_days)) OR NEW.start_date
        BETWEEN B1.start_date AND (SELECT get_end_date(B1.start_date, B1.number_of_days)))
        LOOP
                UPDATE bid SET status = 'Rejected'
                WHERE ct_email = bid_row.ct_email
                AND start_date = bid_row.start_date
                AND po_email = bid_row.po_email
                AND pet_name = bid_row.pet_name;
        END LOOP;
        END IF;
        FOR bid_row IN SELECT * FROM bid B2 WHERE B2.status = 'Pending'
        AND B2.po_email = NEW.po_email AND B2.pet_name = NEW.pet_name
        AND (NEW.start_date BETWEEN B2.start_date AND (SELECT get_end_date(B2.start_date,
        B2.number_of_days))
        OR B2.start_date BETWEEN NEW.start_date AND (SELECT get_end_date(NEW.start_date,
        NEW.number_of_days)))
        LOOP
                UPDATE bid SET status = 'Rejected'
                WHERE ct_email = bid_row.ct_email
                AND start_date = bid_row.start_date
                AND po_email = bid_row.po_email
                AND pet_name = bid_row.pet_name;
        END LOOP;
RETURN NEW;
END; $$
LANGUAGE 'plpgsql';

CREATE TRIGGER on_accept_trigger
   BEFORE INSERT
   ON bid
   FOR EACH ROW
   EXECUTE PROCEDURE check_on_bid_accept();
```

3. **Notification Trigger on Bid Acceptance**

This trigger was written for notifying the pet owner if a caretaker has accepted his bid. Whenever a row in the bid table is updated, this notification will be sent over a channel which our backend is listening to. After the backend receives it, it will be sent over to the pet owner who is currently logged in.

```
CREATE OR REPLACE FUNCTION bid_accepted_notify()
   RETURNS TRIGGER
AS $$
BEGIN
        IF NEW.status = 'Accepted' THEN
                PERFORM pg_notify('accepted_channel', row_to_json(NEW)::text);
```

```
        END IF;
RETURN NEW;
END; $$
LANGUAGE 'plpgsql';

CREATE TRIGGER accept_bid_trigger_notify
   AFTER UPDATE
   ON bid
   FOR EACH ROW
   EXECUTE PROCEDURE public.bid_accepted_notify();
```

Complex Queries

1. **Searching of Caretakers who are are available to take care of all the Pet Owner's Pets**

This query allows Pet Owners, who own multiple kinds of pets (e.g. a cat and a dog) to search for a particular Caretaker who is able to take care of all of his/her pets during the requested dates.

The Caretakers are returned if they fulfil the following criteria:

1. Must be a full-time Caretaker (since only full-timer can auto-accept jobs)
2. Must not be on leave during the requested time period
3. Must be under Caretaker's selected pet category
4. Must not take care of more than 5 pets during the requested time period

```
CREATE OR REPLACE FUNCTION search_care_takers_all_category(startdate VARCHAR, enddate
VARCHAR, poemail VARCHAR)
        RETURNS TABLE(email VARCHAR, name VARCHAR, address VARCHAR, user_role TEXT,
ratings NUMERIC, region VARCHAR, reg_date TIMESTAMP)
AS $$ BEGIN
RETURN QUERY SELECT * FROM (
        SELECT DISTINCT PU.email, PU.name, PU.address, 'Full-Timer', (SELECT get_avg_ratings
        (PU.email)) AS ratings, PU.region, PU.reg_date
        FROM full_timer FT NATURAL JOIN care_taker CT NATURAL JOIN pcs_user PU
        WHERE FT.email NOT IN (SELECT DISTINCT FL.email FROM ft_leaves FL WHERE FL.date >=
                                        TO_DATE(startdate, 'YYYY-MM-DD')
        AND FL.date <= TO_DATE(enddate, 'YYYY-MM-DD'))
        AND (SELECT COUNT(*) FROM bid B WHERE B.ct_email = PU.email
                AND ((B.start_date BETWEEN TO_DATE(startdate, 'YYYY-MM-DD')
                AND TO_DATE(enddate, 'YYYY-MM-DD'))
                OR (TO_DATE(startdate, 'YYYY-MM-DD') BETWEEN B.start_date
                AND (SELECT get_end_date(B.start_date, B.number_of_days))))) < 5
        AND NOT EXISTS (SELECT 1 FROM pet_owns PO WHERE PO.po_email = poemail
        AND NOT EXISTS (SELECT 1 FROM kind_of_pets KP WHERE KP.ct_email = FT.email
                                AND KP.category = PO.category))
        GROUP BY PU.email
        UNION
        SELECT DISTINCT PU.email, PU.name, PU.address, 'Part-Timer', (SELECT get_avg_ratings
        (PU.email)) AS ratings, PU.region, PU.reg_date
```

```
        FROM part_timer PT NATURAL JOIN care_taker CT NATURAL JOIN pcs_user PU
        WHERE PT.email IN (SELECT DISTINCT PA.email FROM pt_availability PA
                        WHERE PA.start_date <= TO_DATE(startdate, 'YYYY-MM-DD')
                        AND PA.end_date >= TO_DATE(enddate, 'YYYY-MM-DD'))
        AND (SELECT COUNT(*) FROM bid B WHERE B.ct_email = PU.email
                AND ((B.start_date BETWEEN TO_DATE(startdate, 'YYYY-MM-DD')
                AND TO_DATE(enddate, 'YYYY-MM-DD'))
                OR (TO_DATE(startdate, 'YYYY-MM-DD') BETWEEN B.start_date
                AND (SELECT get_end_date(B.start_date, B.number_of_days)))))
                < (CASE WHEN (SELECT get_avg_ratings (PU.email)) >= 4 THEN 5 ELSE 2 END)
        AND NOT EXISTS (SELECT 1 FROM pet_owns PO WHERE PO.po_email = poemail
        AND NOT EXISTS (SELECT 1 FROM kind_of_pets KP WHERE KP.ct_email = PT.email
                                AND KP.category = PO.category))
        GROUP BY PU.email) AS q
        ORDER BY (CASE WHEN q.region = (SELECT PU.region FROM pet_owner PO NATURAL
        JOIN pcs_user PU WHERE PO.email = poemail) THEN 1 ELSE 0 END) DESC, q.ratings DESC;
END; $$
LANGUAGE 'plpgsql';
```

2. **Query to retrieve the Star Performer Caretaker**

This allows the administrators and pet owners to retrieve the star performer of the previous month caretaker for a particular category. Star Performers will be given the spotlight on our website at the same time. A caretaker is deemed to be a star performer if he/she meets the following criteria:

1. Take care of a particular pet category for 5 times for the past month
2. Obtained average ratings of 4
3. Did not have any cancellation for the past month

```
CREATE OR REPLACE FUNCTION get_star_performer_ct(s_category VARCHAR)
   RETURNS TABLE(email VARCHAR, name VARCHAR, address VARCHAR, user_role TEXT, ratings
NUMERIC, region VARCHAR, reg_date TIMESTAMP)
AS $$ BEGIN
        RETURN QUERY SELECT PU.email, PU.name, PU.address, 'Full-Timer', AVG(rating) AS
        ratings, PU.region, PU.reg_date FROM full_timer FT
        NATURAL JOIN care_taker NATURAL JOIN pcs_user PU
        INNER JOIN bid B ON FT.email = B.ct_email
        WHERE EXTRACT(MONTH FROM B.start_date) = EXTRACT(MONTH FROM current_date) - 1
        AND (SELECT category FROM pet_owns PO WHERE PO.name = B.pet_name
        AND B.po_email = PO.po_email) = s_category
        AND status = 'Accepted'
        AND NOT EXISTS (SELECT 1 FROM bid B1 WHERE B1.ct_email = FT.email
                                AND status = 'Cancelled' AND EXTRACT(MONTH FROM B1.start_date) =
                                EXTRACT(MONTH FROM current_date) - 1)
        GROUP BY PU.email, PU.name
        HAVING COUNT(*) >= 5 AND AVG(rating) >= 4
        UNION
        SELECT PU.email, PU.name, PU.address, 'Part-Timer', AVG(rating) AS avg_rating, PU.region,
        PU.reg_date FROM part_timer PT NATURAL JOIN care_taker NATURAL JOIN pcs_user PU
```

```
        INNER JOIN bid B ON PT.email = B.ct_email
        WHERE EXTRACT(MONTH FROM B.start_date) = EXTRACT(MONTH FROM current_date) - 1
        AND (SELECT category FROM pet_owns PO WHERE PO.name = B.pet_name
                AND B.po_email = PO.po_email) = s_category
        AND status = 'Accepted'
        AND NOT EXISTS (SELECT 1 FROM bid B1 WHERE B1.ct_email = PT.email
                        AND status = 'Cancelled' AND EXTRACT(MONTH FROM B1.start_date) =
                        EXTRACT(MONTH FROM current_date) - 1)
        GROUP BY PU.email, PU.name
        HAVING COUNT(*) >= 5 AND AVG(rating) >= 4;
END; $$
LANGUAGE 'plpgsql';
```

3. **Query to retrieve the salary of caretaker**

This allows the caretakers to view their total workday of taking care of pets, total salary, and bonuses.

1. A full-time caretaker is given a bonus if he/she has worked more than 60 pet-days. The caretaker will receive 80% of the price on each excess pet-day.
2. A part-time caretaker always gets 75% of their stated price.

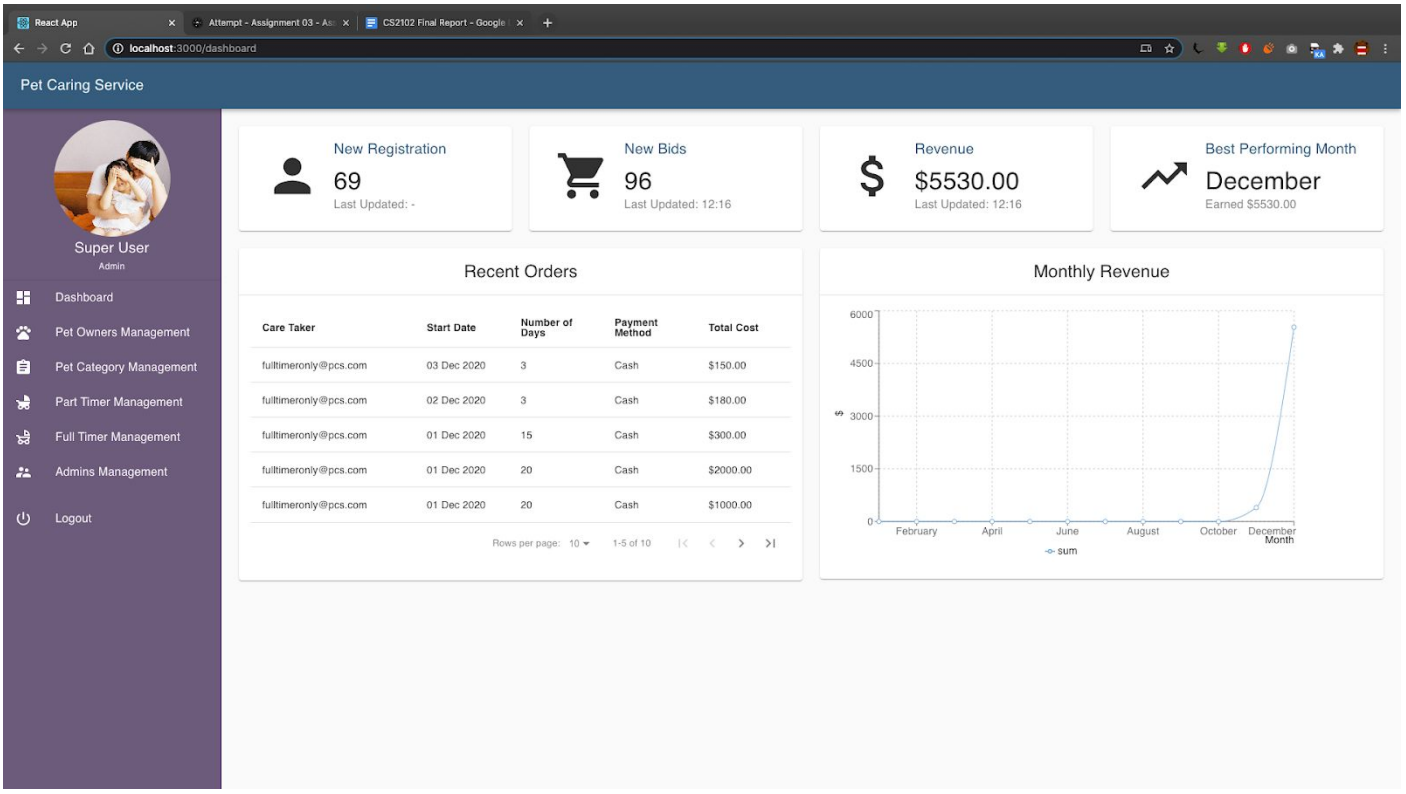
```
CREATE OR REPLACE FUNCTION get_salary(email VARCHAR, selected_month NUMERIC)
   RETURNS TABLE(work_days BIGINT, salary NUMERIC, bonus NUMERIC)
AS $$
DECLARE
        total_days INTEGER;
        tuple RECORD;
        bonus_days INTEGER = 60;
        extra_money NUMERIC(10,2) = 0;
        total_salary INTEGER = 3000;
BEGIN
        SELECT SUM(number_of_days) FROM bid WHERE ct_email = email AND status = 'Accepted'
AND EXTRACT(month FROM start_date) = selected_month INTO total_days;
        IF (SELECT COUNT(*) FROM full_timer FT WHERE FT.email = user_email) = 1 THEN
                IF total_days > 60 THEN
                        FOR tuple IN
                                SELECT *
                                FROM bid
                                WHERE ct_email = user_email
                                AND status = 'Accepted'
                                AND EXTRACT(month FROM start_date) = selected_month
                                ORDER BY start_date ASC, total_cost DESC
                        LOOP
                                IF bonus_days >= tuple.number_of_days THEN
                                        bonus_days = bonus_days - tuple.number_of_days;
                                ELSE
                                        extra_money = extra_money + ((tuple.total_cost /
tuple.number_of_days) * (tuple.number_of_days - bonus_days) * 0.8);
                                        bonus_days = 0;
```

```
                    END IF;
                END LOOP;
                total_salary = total_salary + extra_money;
            END IF;
        ELSEIF (SELECT COUNT(*) FROM part_timer PT WHERE PT.email = user_email) = 1 THEN
            SELECT SUM(total_cost) AS total FROM bid WHERE ct_email = user_email INTO tuple;
            total_salary = tuple.total * 0.75;
        ELSE
            RAISE EXCEPTION 'Invalid Care Taker Found';
            RETURN;
        END IF;
    RETURN QUERY
            SELECT COALESCE(SUM(B.number_of_days), 0),
                total_salary,
                extra_money
            FROM bid B
            WHERE B.ct_email = user_email
                AND status = 'Accepted'
                AND EXTRACT(month FROM start_date) = selected_month;
END; $$
LANGUAGE 'plpgsql'
```

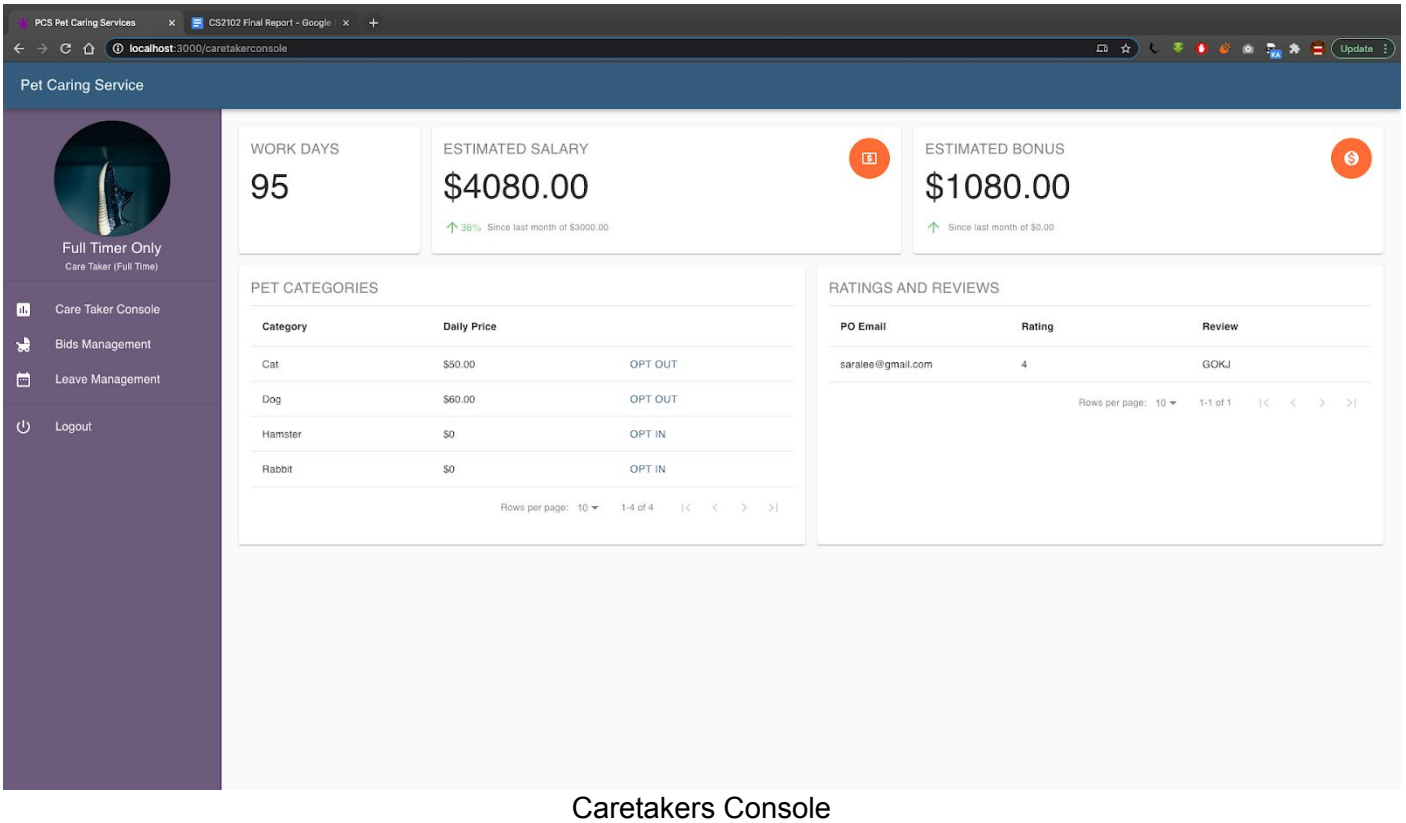
Project Specification & Software Tools

Components	Framework / Libraries / Software Tools
Database	PostgreSQL (hosted on AWS RDS) pgcrypto pgAdmin 4
Backend	Node.js Express.js pg-promise pg-listen
Frontend	React.js Material-UI Heroku
Software Tools	Git / Github

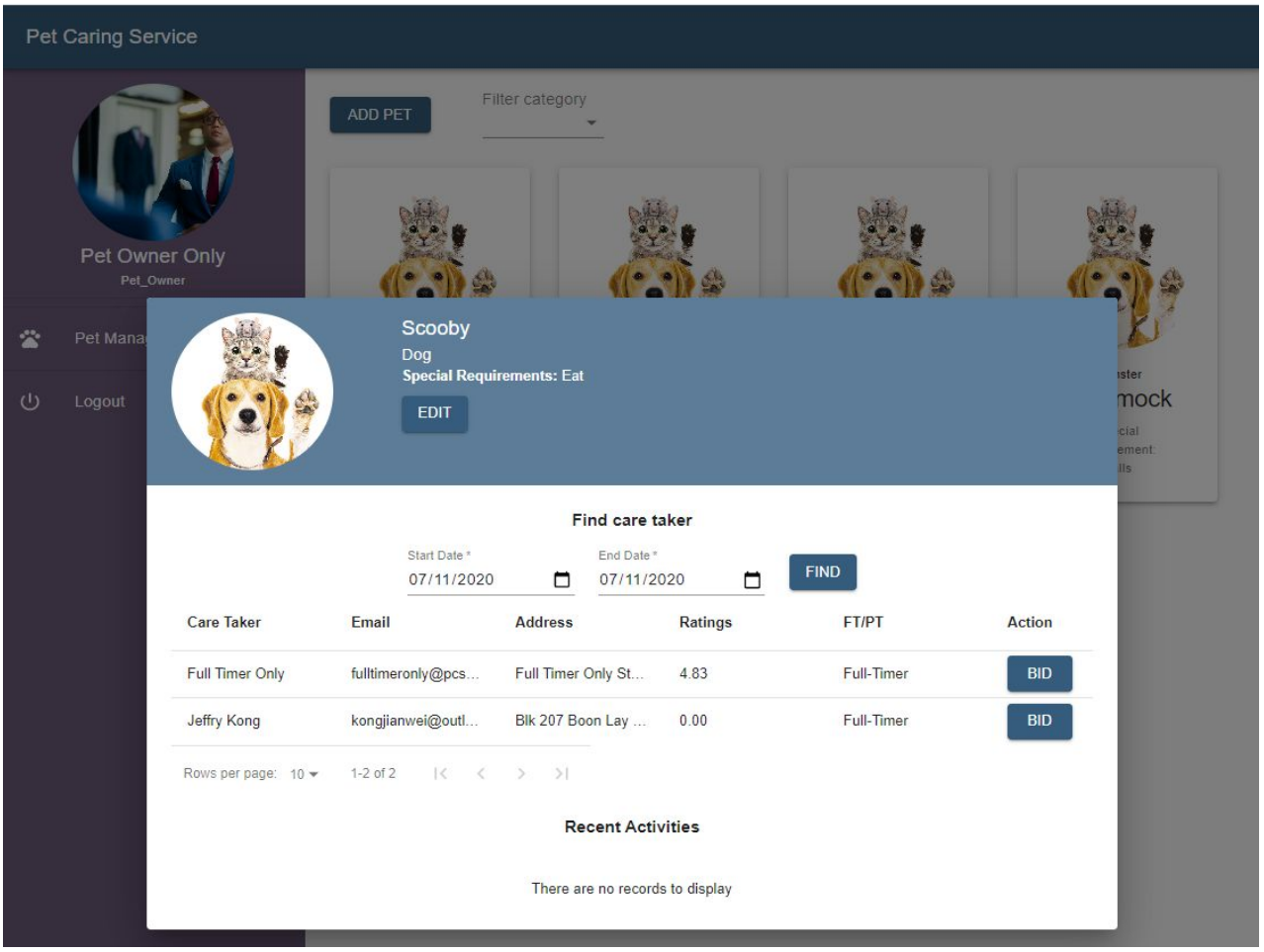
Screenshots



Admin Console



Caretakers Console



Pet Owner Console

Summary

Throughout this project, we realized that triggers are useful for enforcing constraints that can't be expressed in a table. They can also provide useful functions like notify and ensure that even if an invalid request is somehow sent to the database, the data will not reach an undesirable state.
Additionally, as we have learnt about normal forms, the BCNF and 3NF makes the data schema very clean and ensures no unnecessary duplication.

A few of the challenges we faced were on web development, as some of us were new to the React Framework and this provided a good learning opportunity in modern frameworks. For those of us using pgAdmin for the first time, there was a slight learning curve with regards to interacting with the UI. This gradually got better as we started to get accustomed to it.

Overall, it's a good learning experience as databases are crucial to learn and may play a big role in our future careers.