CS2102 Project Report
Team 36

1. AND 2. Project responsibilities

Name	Matric No.	Responsibilities
Fong Yong Jie	A0183816X	Frontend, Quality Check, Report, ER Diagram
Li Zi Ying	A0189497B	Backend, Report, ER Diagram
Ng Ming Liang	A0183324J	Frontend, Report, ER Diagram
Pang Jia Da	A0184458R	Backend, Report, ER Diagram
Pang Jia Hao	A0160717H	Backend, Report, ER Diagram

3. Data Requirements and Functionalities

3.1. Description of application

Our application is called Pet Care Center. Its main function is a platform for pet owners to find caretakers to look after their pets.

Pet owners can use this application to find caretakers to look after their pets. They can filter from a set of criteria in order to find caretakers that best suit their needs. They can also ask the system to recommend some caretakers which the system has identified are a good match for them. Users of this application can also sign up as part time caretakers.

Bidding system: Outstanding caretakers usually have high demand for their service, and the demand can be larger than what the caretaker is able to handle. To resolve this, the users of this application can use the bidding system. The bidding system allows pet owners to compete for the services of their favourite caretakers.

The **rating system** of the application allows pet owners to rate the services of a caretaker that they have hired, and allows future users to determine which caretakers are good and bad.

There is also a forum for all Users to post and comment. This can help caretakers to get recognition on the forum and be more popular for pet owners to bid for them. Also for users to interact and find answers to their pet problems.

Entity	Functionality
Users	1. Create, edit and delete a new forum post 2. Create and delete comments on forum post 3. Manage own profile 4. Read a forum post and its comments 5. Read/View a caretaker's profile or availability a. Only appears if caretaker has availability in the next 2 years 6. Read/View reviews made on caretaker 7. Update own profile 8. Delete a comment made by user himself

	9. Filter for caretakers based on some criteria a. Name b. Full Time / Part Time c. Available Date Between Specified Start and End Dates d. Pet Type e. Minimum and Maximum Price f. Minimum Rating
Admin	1. Create new admin / full time caretaker 2. Create new pet type with its base price 3. Read a User profile 4. Read a Caretaker detailed information a. Salary for this month b. Amount of working days this month c. Profit earned from the Caretaker (Revenue - Salary) 5. Update the base price of a pet type 6. Delete an existing User a. Admin b. Pet Owner c. Caretaker 7. Delete an existing pet type
Caretakers (Both)	1. Create/Add a new ability to take care of pet type 2. Read all bids to the caretaker a. All b. Pending c. Confirmed d. Done 3. Update/Accept/Reject a bid made by a petowner to caretaker 4. Update a bid to paid 5. Delete ability to take care of pet type 6. View a calendar with successful bids a. Bids on the calendar can be clicked to see the information of the successful bid b. For easy tracking c. Number of working days in the month d. Salary they got for that month
Caretaker (Full Time)	1. Create/Apply for leave dates 2. View leave dates on calendar
Caretaker (Part Time)	1. Create/Apply for available dates 2. Update daily price of taking care of pet type 3. View availability dates on calendar
Pet Owners	1. Create a new pet 2. Create/Place a bid for a caretaker 3. Read all bids made by pet owner a. All b. Pending c. Confirmed d. Done 4. Update details of existing pet a. Name

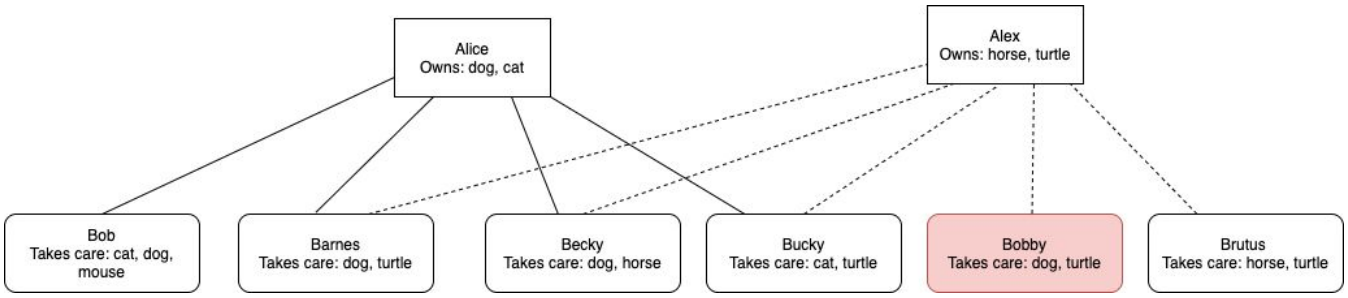
	b. Description c. Special Requirements d. Pet Type 5. Delete an existing pet 6. View a calendar with successful bids a. Bids on the calendar can be clicked to see the information of the successful bid b. For easy tracking 7. Get recommendation for caretakers based on previous transactions

3.2. Interesting Features

Recommendation function

A petowner can request our application to recommend them a list of caretakers. We say that a Pet owner *likes* a caretaker if the caretaker has worked for the pet owner at least once, and the pet owner's average rating of the caretaker is at least 4. We consider two Pet owners to be *similar* if there exist at least three caretakers who both pet owners like. Let A be a pet owner. The recommendation function will find all caretakers C who have never worked for A (never had a confirmed bid for them by A), and are able to take care of some pet owned by A, and there exists some pet owner B such that A is similar to B, and B likes C.

Example:



Suppose Alice likes Bob, Barnes, Becky and Bucky. Suppose Alex likes Barnes, Becky, Bucky, Bobby and Brutus. Since they both like Barnes, Becky and Bucky, Alice and Alex are similar. Now, suppose Alice has never hired either Bobby or Brutus. Bobby can take care of Alice's Dog, while Brutus is unable to take care of any of Alice's pets. Thus, Bobby will be recommended to Alice.

Filter function

As there are many caretakers associated with PCS, pet owners need to be able to find the caretakers who are most suitable for them. This is achieved through a comprehensive filtering feature. PCS is able to filter by:

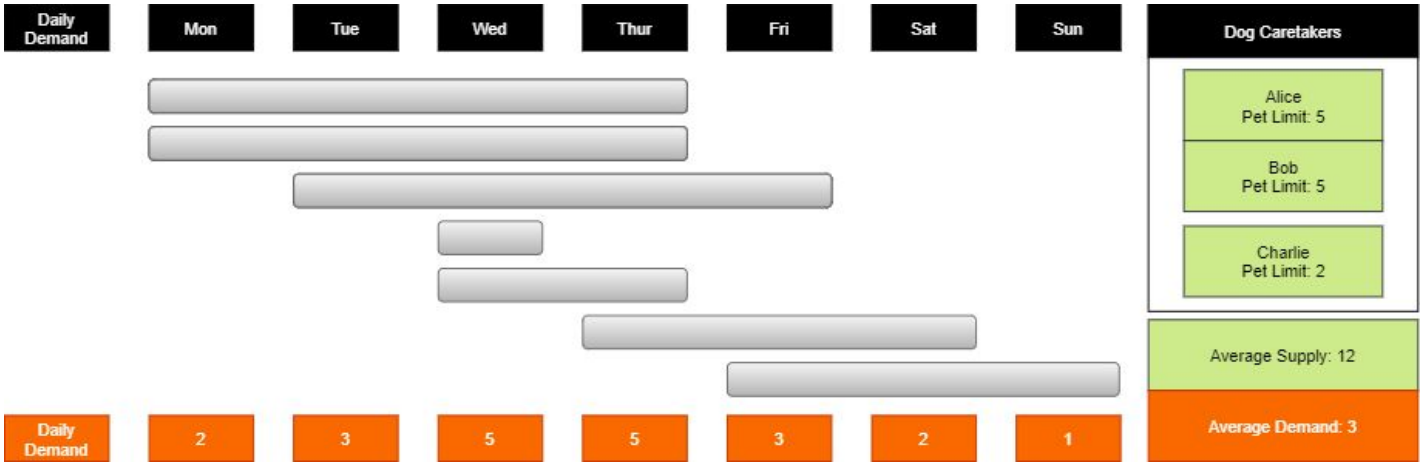
- Caretaker name. The pet owner can specify a string, and only caretakers whose name contains this substring will be included.
- Caretaker rating. The pet owner can specify the minimum rating, and only caretakers with an equal or higher rating will be included.
- Caretaker Type. Only the specified type of caretaker (full time or part time) will be included.
- Caretaker availability. The pet owner may be searching for a caretaker who is able to work on a particular interval. The pet owner can supply an interval, and only caretakers able to work on this interval will be included.
- Pet type. Only caretakers who are qualified to take care of this pet type will be included.
- Minimum and maximum price. If a pet type is specified, the pet owner can further specify a price range. Only caretakers whose daily price for the specified pet type falls into this price range will be included.

Summary statistics

Our application allows PCS Admins to find useful statistics of the caretakers and pet owners.

- View salary to be paid to each caretaker for services completed in previous month
    - The revenue earned from a job, and their pet days are only considered in full when they have completed an entire job.
    - Hence the jobs that are considered in a month's salary calculation are those that were completed during that month.The total revenue a caretaker earns is the sum of revenue from all jobs ending during that query period. The revenue from a single job is the number of pet days worked times the amount of money bidded per pet day.
    - For part time caretakers, their salary is 0.75 of their total revenue.
    - For full time caretakers, if their total pet days worked the previous month is less than 60, they will be paid $3000 base salary
    - If a caretaker works more than 60 pet days, they will receive, in addition to their base $3k salary, 80% of the revenue earned by the pet days in excess of 60.
    - This is calculated using an average price per pet day which is their total revenue divided by total pet days.
- View caretaker revenue
    - PCS Admins can view the number of pet days worked and the amount of revenue
    - This function allows PCS Admins to determine which are their best or worst performing caretakers.
- View demand and supply for a specified pet type.
    - The demand for a pet type on a specified day is computed as the number of bids for that pet type that have an interval containing the specified day. The average demand for a pet type on a specified interval is the average of the demand over all days in the specified interval.
    - The average supply of a pet type is the sum of the caretaker's pet limit over all caretakers that can care for the pet type. The supply of a pet can be seen as the theoretical maximum number of pets of that pet type that PCS can care for every day.
    - By comparing the demand and supply for a particular type of pet per month, PCS can determine if more caretakers need to be hired or trained to care for that pet type.

Example.



Consider the above set of bids over a one week interval. The gray boxes are bids for pet type Dog. We can see that the demand on Monday is 2, demand on Tuesday is 3, etc.The average demand for this week is 3. Suppose PCS has three caretakers Bob, Bucky and Becky. The sum of their pet limits is 12. Hence, the average supply of Dog caretakers is 12.

3.3. Data constraints

We make the following assumptions about the data inputs to our application.

User

- All users are required to have registered an account before logging in.
    - This is to replicate real-world applications, where accounts must be created before being allowed to interact with other users of the application.
- A user can only sign up as a pet owner or a part time caretaker. If a user wants to be a full time caretaker, their accounts can only be created by the PCS admins.
- New admins can also only be created by PCS admins.
- We assume that emails identify a user across time. For example, if a user registers with her email alice@gmail.com, uses the app, is deleted from the app, and subsequently another user signs up using the email alice@gmail.com again, we will assume that both users are the same person.

Pet owner

- A pet owner never names two of his pets with the same name, even if he does not own both pets simultaneously. The reason for the assumption is to allow PCS to accurately store bids in the BidsFor table. For example, suppose a user makes a bid for a Dog named Roger, and subsequently deletes Roger, adds in a new Dog called Roger, and makes a bid for New Roger. Without this assumption, if we want to find all bids for Old Roger, we would not be able to do so, because there is no way to distinguish Old Roger and New Roger.
- A pet owner does not review and rate a job before the job has been fully completed. This does not affect the application, but we include it for real-world sensibilities.

Caretaker

- We assume that a caretaker taking leave will take a full day leave, and a caretaker indicating availability means a full day availability. We also assume that the caretaker's jobs are for one or more consecutive full days. Thus the dates of caretaker leave, availability, and jobs are stored as DATE data type as the exact time does not matter.
- A caretaker is not allowed to change from part time to full time or from full time to part time.

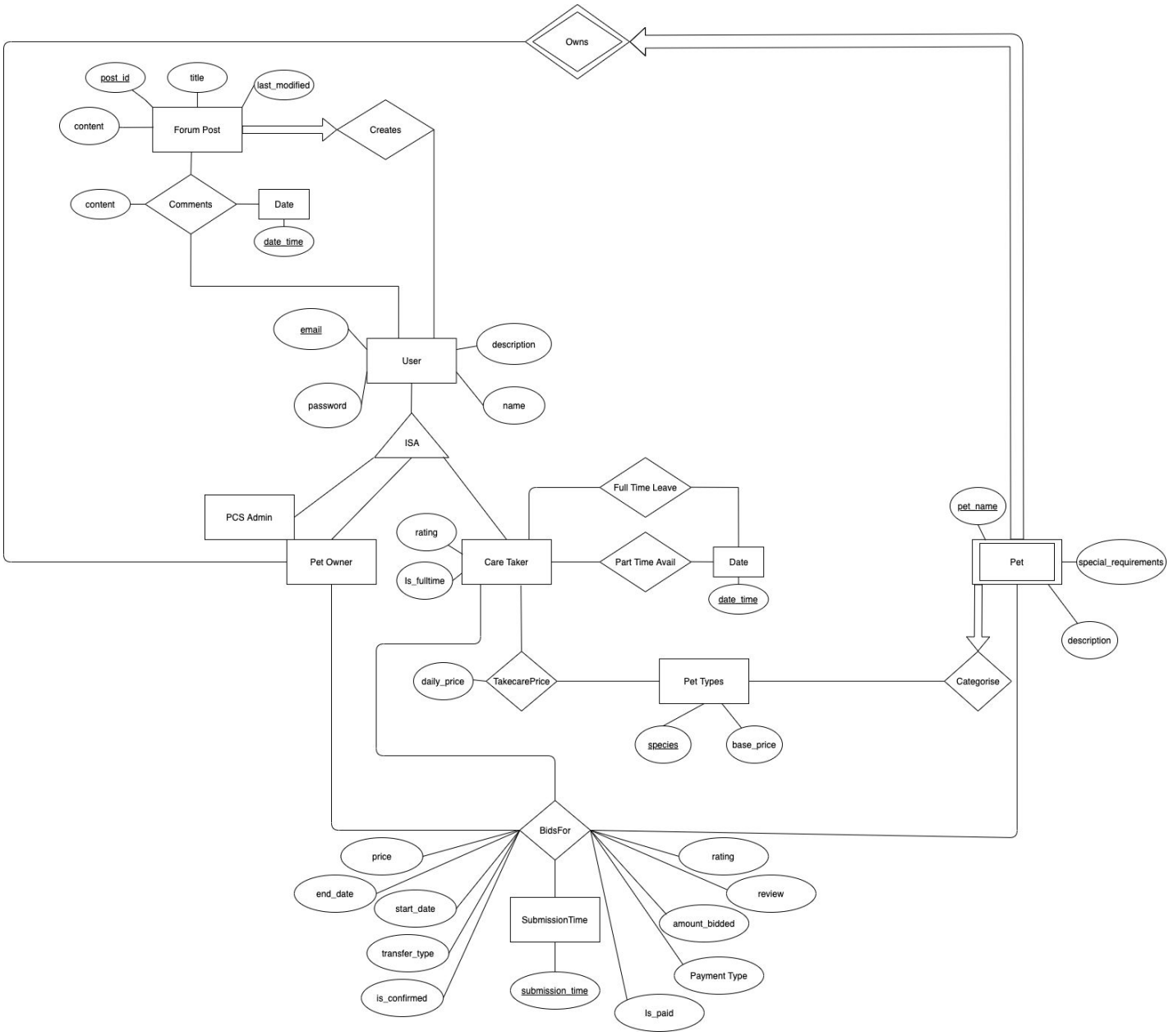
Bids

- We assume that a single pet owner can create bids at the maximum speed of one per microsecond. Thus, we can identify a unique bid from the four pieces of data: the pet owner who places the bid, the caretaker for whom the bid is placed, the pet's name, and the submission time to microsecond precision.
- We assume that pet owners never need to edit or cancel a bid.
- We assume that once a bid is confirmed, there is no cancellation of the job.
- We assume that payment is made only after a bid is confirmed.
- We assume it is the caretaker that indicates that the bid is paid.

Forum (Posts and Comments)

- Justification for serial id for Post: Support deletion of a user account who made the post and keeping his Posts. If the user deletes his account, we do not want a helpful post with many comments that was created by him to be deleted as well. Also, posts with the same title can be created several years in the future, which makes sense because info can become outdated. We did not implement serial id for comments as our intention is to remove all comments from a deleted User.
- We assume that each User can create comments at the maximum speed of one per microsecond. Thus, we can identify a unique comment from the three pieces of data: the post_id, the comment author's email and the time of comment to microsecond precision.

4. ER Diagram



Justifications for any non trivial design decisions

- Initially, the availability/leave of a caretaker is captured using a bitmap, where we have a bitmap that can represent 2 years. However, we realised that it was really hard to constantly readjust the bitmap such that the caretaker can always declare availability or leave dates 2 years in advance. We then decided to capture it using entities FullTimeLeave and PartTimeAvail. In this way, previous records of the leave and availability can also be kept.

4.1. Application constraints not captured by ER Diagram

Caretaker

- The ER diagram does not reflect the limit on the number of pets a caretaker can care for simultaneously. This limit is 5 for full time caretakers, 5 for part time caretakers with a good rating, and 2 for part time caretakers without a good rating. Here, we define a *good* rating as a rating that is not less than 4.
- Does not reflect several leave/availability related constraints
    - Full time caretaker cannot take leave on a day which he has a confirmed job
    - Part time caretaker cannot remove availability when he has confirmed bid on it
    - A bid cannot be submitted for a part time caretaker on an interval where the caretaker has not made himself available.
- Does not reflect the relationship between the rating of a caretaker and the entries of the BidsFor table. In our application, the rating of the caretaker is determined as the average rating of all the jobs for which he has received a rating.

Bids

- Does not reflect that the bidsFor relation has a pseudo foreign key: At the point of creation of a bid, (petowner, pet_name) references Pets(email, pet_name). This is a foreign key because a pet owner can only make a bid for a pet that he owns. However, if the pet is deleted, the referencing entry in the BidsFor table is not modified (not deleted or set to null). This is because the BidsFor table is intended to store all bids ever created, for business analysis reasons.
- Does not reflect that a User who is both a pet owner and caretaker is not permitted to submit a bid for himself.
- Does not reflect that for a bid, the caretaker should be able to take care of the pet. (This is enforced on frontend)
- Does not reflect that for a bid, the amount bidded is not less than the daily price charged by the caretaker for that pet type.
- Does not reflect that if a bid is for a part time caretaker, the interval of the bid must be indicated by the part time caretaker as available

5. Table Schema

Under explanations, we explain any stuff that we are doing / conventions we are using, e.g. null = pending, true = accepted, etc..

In the next section, we explain stuff that is not captured by schema

Table	Explanations
``CREATE TABLE Users (name VARCHAR(30) NOT NULL, email VARCHAR(30) PRIMARY KEY, description VARCHAR(255), password VARCHAR(60) NOT NULL); CREATE TABLE Caretakers (email VARCHAR(30) PRIMARY KEY REFERENCES Users(email) ON DELETE CASCADE, is_fulltime BOOLEAN NOT NULL, rating DECIMAL(10, 2), CHECK (0 <= rating AND rating <= 5)); CREATE TABLE PetOwners (email VARCHAR(30) PRIMARY KEY REFERENCES Users(email) ON DELETE CASCADE```	Tables for the Users and the three main types The rating of a Caretaker is initially null (for a new caretaker who has not completed any jobs). When a caretaker has one or more completed jobs that have been rated, his rating will then be the average rating over all his jobs

``); CREATE TABLE PcsAdmins (email VARCHAR(30) PRIMARY KEY REFERENCES Users(email) ON DELETE CASCADE); ```	
`` CREATE TABLE PartTimeAvail (email VARCHAR(30) REFERENCES Caretakers(email) ON DELETE CASCADE, work_date DATE, PRIMARY KEY (email, work_date)); CREATE TABLE FullTimeLeave (email VARCHAR(30) REFERENCES Caretakers(email) ON DELETE CASCADE, leave_date DATE NOT NULL, PRIMARY KEY (email, leave_date)); ```	Tables for tracking Caretaker leave and availability. This table stores all leave that is planned, as well as leave that has been taken (in the past).
`` CREATE TABLE PetTypes (species VARCHAR(30) PRIMARY KEY NOT NULL, base_price DECIMAL(10,2) NOT NULL); CREATE TABLE Pets (email VARCHAR(30) REFERENCES PetOwners(email) ON DELETE CASCADE, pet_name VARCHAR(30), special_requirements VARCHAR(255), description VARCHAR(255), species VARCHAR(30) REFERENCES PetTypes(species) ON DELETE CASCADE, PRIMARY KEY (pet_name, email)); ```	Tables to show pet related information. The PetTypes table stores all the types of pets recognized by the app. Each pet type has an associated base price that all full time caretakers will charge for taking care of that pet type. The Pets table stores what Pets are owned by what Pet owners. As mentioned in data constraints, we assume that Pet owners do not have pets with the same names.
`` CREATE TABLE TakecarePrice (email varchar(30) REFERENCES Caretakers(email) ON DELETE CASCADE, species varchar(30) REFERENCES PetTypes(species), daily_price DECIMAL(10,2) NOT NULL, PRIMARY KEY (email, species)); ```	The TakecarePrice table stores the types of pet that each caretaker is able to take care of, as well as the daily price that the caretaker charges for that type of pet. For full time caretakers, the daily price is computed as base price + 5 * rating. Since rating is nonnegative, this ensures the daily price is not less than the base price. For part time caretakers, the daily price is whatever they want to set it as. Their daily price is not affected by any triggers.
`` CREATE TABLE BidsFor (owner_email VARCHAR(30), caretaker_email VARCHAR(30) REFERENCES CareTakers(email) ON DELETE CASCADE, pet_name VARCHAR(30), submission_time TIMESTAMP, start_date DATE, ```	The BidsFor table is designed to record all the bids and jobs that have been handled in PCS. The bids here are used to calculate the amount of traffic on PCS and Caretaker salary. There are reasons why many of the table

<pre> end_date DATE, price DECIMAL(10,2), amount_bidded DECIMAL(10,2), is_confirmed BOOLEAN DEFAULT NULL, is_paid BOOLEAN DEFAULT False, payment_type payment_type, transfer_type transfer_type, rating DECIMAL(10, 1) DEFAULT NULL CHECK (rating ISNULL OR (rating >= 0 AND rating <= 5)), review VARCHAR(255) DEFAULT NULL, PRIMARY KEY (caretaker_email, owner_email, pet_name, submission_time) CONSTRAINT bidsfor_dates_check CHECK (submission_time < start_date AND start_date <= end_date), CONSTRAINT bidsfor_price_le_bid_amount CHECK (price <= amount_bidded), CONSTRAINT bidsfor_confirm_before_paid CHECK (NOT is_paid OR is_confirmed));</pre>	columns do not reference other table columns. For example, owner_email or pet_name. This is purposely such that if the pet owner or pet is deleted and the bid is deleted, the caretaker would no longer be paid for that bid. Whereas if the caretaker is deleted, then there would not be a need to calculate his salary so delete on cascade is used. When a bid is submitted, the is_confirmed attribute is initially null. This indicates that the bid is pending. The is_paid attribute is initially false. It is updated to true when the Pet owner makes payment. The is_paid attribute cannot be set to true while is_confirmed is null or false.
<pre>CREATE TABLE Posts (post_id SERIAL PRIMARY KEY, email VARCHAR(30) NOT NULL REFERENCES Users(email) ON DELETE CASCADE, title VARCHAR(255), cont TEXT, last_modified TIMESTAMP DEFAULT NOW()); CREATE TABLE Comments (post_id INTEGER REFERENCES Posts(post_id) ON DELETE CASCADE, email VARCHAR(30) REFERENCES Users(email) ON DELETE CASCADE, date_time TIMESTAMP DEFAULT NOW(), cont TEXT, PRIMARY KEY (post_id, email, date_time));</pre>	We chose the serial primary key for the posts as we want to keep the entry if the user created the post is deleted. The post might contain essential information. Other users might put in effort in their comments to the post and feel unhappy that it is deleted. For the case of comments, since the user is deleted, he most likely will not need/want to access his comments anymore.

Constraints not captured by the schema

Users

- A User must be either a Petowner, Caretaker, or PCSAdmin. This is enforced with triggers.
    - Inserting a new User/Petowner/Caretaker is done through using a transaction, in order to not be prevented by the trigger.
- A User may be a Caretaker and a Petowner but cannot be both a Caretaker and PCS Admin. They also cannot be Petowner and PCS Admin at the same time. This constraint is enforced with a trigger.
- When there is an update to the rating of a bid, a trigger will recalculate the rating for the corresponding caretaker.

Caretaker

- The schema does not enforce the pet limit constraint.
- A trigger is used to prevent full time caretakers being inserted into PartTimeAvail.
- A trigger is used to prevent a full time caretaker taking leave on a date where he has a confirmed bid.
- A trigger is used to prevent the deletion of availability on a date that the part time caretaker has a confirmed bid.
- There are two triggers to ensure that the daily price is up to date. The first trigger updates the daily price when

there is a change to the base price of a pet type. The second trigger updates the daily price whenever there is a change to a caretakers rating.

- It is possible that a Caretaker circumvents the 2x150 consecutive day rule, by deleting his leave after he has taken it, and then re-applying for leave. We considered adding a trigger to prevent deletion of leave in the past. However, this would also make it hard for an authorized entity such as the PCS Admin to delete leave. Hence, we decided not to implement this trigger.

Bids

- Does not reflect that a User who is both a pet owner and caretaker is not permitted to submit a bid for himself. This is enforced by a trigger.
- Does not reflect that for a bid, the caretaker should be able to take care of the pet. This is enforced by our front end.
- The price attribute in a bid equals the daily price of the caretaker at the time when the bid was created. This is enforced by automatically setting the price attribute to the caretakers daily price when a bid is created. We record this price in the BidsFor table for historical records because the daily price of caretaker can change as time passes by.
- A full time caretaker should accept the bid once it has been placed, if possible. That is, the caretaker is not on leave during that period, and accepting the bid will not cause the caretaker to exceed his pet limit. This is enforced by a trigger.
- When the current date is past the start date of the bid, the bid is rejected. This is indicated by changing the is_confirmed attribute to false. Ideally, this would be implemented as a trigger, but PSQL does not support time based triggers. Instead, a function is created to reject all bids before the current date. We assume that this function is called every once in a while by a PCS admin.
- The pet owner is able to review the job, and assign a rating to it. A possible trigger would be a trigger to prevent reviews and ratings from being made before completion of the job. However, this trigger seems unnecessary.

6. Functional Dependencies Justifications

Tables	BCNF / 3NF
Admin (Users, PcsAdmins)	BCNF
Caretaker (Users, Caretakers, PartTimeAvail, FullTimeLeave, PetTypes, TakecarePrice)	BCNF
Pet Owner (Users, PetOwners, PetTypes, Pets)	BCNF
Bids (Users, Caretakers, PetOwners, BidsFor, PartTimeAvail, FullTimeAvail)	NONE
Forum (Users, Posts, Comments)	NONE

Justification for not BCNF / 3NF:

- Bids:

- To allow deletion of users, we decided to remove some foreign keys such that the entry will not be deleted. This is understandable as we are using the BidsFor table as a historical ledger. A foreign key ensures that the referencing table contains currently valid data from the referenced table. But a historical record only requires that that holds at the point of insertion. For example, if a caretaker takes care of Alice's dog, Roger, and Alice changes her name or Roger's name, we want the caretaker to still be able to understand his bid records and not question who is the new pet owner name/pet name.

- Forum:
  - Similarly, as mentioned above, the reason for not having it in BCNF and 3NF is for deletion. We do not want all the comments to be deleted because the user who wrote the post is deleted.

7. Complex Triggers

Description	Triggers
Trigger to automatically accept bids for a full time caretaker, if it is possible for the caretaker to fulfil that bid.	

```
CREATE OR REPLACE FUNCTION ft_accept_bid() RETURNS TRIGGER
AS $$
BEGIN
   UPDATE BidsFor BF SET
       is_confirmed = true
   WHERE
       BF.caretaker_email = NEW.caretaker_email AND
       BF.owner_email = NEW.owner_email AND
       BF.pet_name = NEW.pet_name AND
       BF.submission_time = NEW.submission_time AND
       EXISTS (select 1 from Caretakers where email = New.caretaker_email and
is_fulltime=true) AND (
           CASE WHEN (select is_fulltime from caretakers where email =
NEW.caretaker_email) THEN
               not exists (
                   select * from FullTimeLeave
                   where
                       email = NEW.caretaker_email and
                       clash(NEW.start_date, NEW.end_date, leave_date)
               )
           ELSE
               not exists (
                   SELECT generate_series(NEW.start_date, NEW.end_date, '1
day'::interval)::date as work_date
                   EXCEPT (select work_date from parttimeavail where email =
NEW.caretaker_email)
               )
           END
       ) AND (
           getPetLimit(NEW.caretaker_email) > ALL (
               select (
                   select COUNT(*)::int from bidsFor BF2
                   where
                       BF2.caretaker_email = NEW.caretaker_email and
                       BF2.is_confirmed = True and
                       clash(BF2.start_date, BF2.end_date, D.work_date, D.work_date)
               ) as num_jobs
               from (select generate_series(NEW.start_date, NEW.end_date, '1
day'::interval)::date as work_date) as D
           )
       )
   RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

	 DROP TRIGGER IF EXISTS ft_accept_bid ON BidsFor; CREATE TRIGGER ft_accept_bid AFTER INSERT ON BidsFor FOR EACH ROW EXECUTE PROCEDURE ft_accept_bid();
Trigger on BidsFor to prevent a part time caretaker accepting a bid when he has not declared that he is available on the entire interval of that bid.	CREATE OR REPLACE FUNCTION block_inserting_bid_part_time() RETURNS trigger language plpgsql as $$ BEGIN IF EXISTS (select 1 from CareTakers where email = NEW.caretaker_email and is_fulltime = false) AND EXISTS (select generate_series(NEW.start_date, NEW.end_date, '1 day'::interval)::date as work_date EXCEPT select work_date from PartTimeAvail where email = NEW.caretaker_email) THEN RAISE EXCEPTION 'Part time worker does not have availability on this date'; END IF; RETURN NEW; END; $$; DROP TRIGGER IF EXISTS trigger_block_inserting_bid_part_time on BidsFor; CREATE TRIGGER trigger_block_inserting_bid_part_time BEFORE INSERT ON BidsFor FOR EACH ROW EXECUTE PROCEDURE block_inserting_bid_part_time();
Trigger to ensure Caretaker has at least 2x150 consecutive working days in each year. The trigger is on the FullTimeLeave table, and prevents insertion of leave dates which would violate this property.	CREATE OR REPLACE FUNCTION isLeaveValidTrigger() RETURNS trigger language plpgsql as $$ BEGIN IF NOT ((select sum(len / 150) from (select (lead(leave_date, 1) over (order by leave_date asc)) - leave_date - 1 as len FROM (select * from fulltimeleave where email = NEW.email and EXTRACT(YEAR FROM leave_date) = EXTRACT(YEAR FROM NEW.leave_date)::int UNION select NEW.email as email, ((EXTRACT(YEAR FROM NEW.leave_date)::int - 1) || '-12-31')::date as leave_date UNION select NEW.email as email, ((EXTRACT(YEAR FROM NEW.leave_date)::int + 1) || '-01-01')::date as leave_date

	``) L1) L2) >= 2) THEN RAISE 'Invalid leave pattern'; END IF; RETURN NEW; END; $$; DROP TRIGGER IF EXISTS is_leave_valid_trigger ON FullTimeLeave; CREATE CONSTRAINT TRIGGER is_leave_valid_trigger AFTER INSERT ON FullTimeLeave FOR EACH ROW EXECUTE PROCEDURE isLeaveValidTrigger(); ```

8. Complex Queries

Description	Queries
PSQL query to find all caretakers to recommend to Alice.	``` select email, name, rating, CASE WHEN is_fulltime THEN 'Full Time' ELSE 'Part Time' END from Users NATURAL JOIN (select * from caretakers CT where exists (select * from petowners PO where isSimilar('alice@gmail.com', PO.email) and PO.email != 'alice@gmail.com' and likes(PO.email, CT.email)) EXCEPT select * from caretakers CT where exists (select 1 from bidsfor where owner_email = 'alice@gmail.com' and caretaker_email = CT.email and is_confirmed = True) EXCEPT select * from caretakers CT where not exists (select species from pets where email = 'alice@gmail.com' INTERSECT select species from Takecareprice TCP where TCP.email = CT.email)) AS Rec; ```

PSQL Function for filtering caretakers.

```
CREATE OR REPLACE FUNCTION filterCaretakers(
   p_ct_name varchar, -- caretakers with this in their name
   p_rating DECIMAL(10, 2), -- caretakers with at least this rating
   p_is_fulltime boolean, -- caretaker of this type
   p_pet_type varchar, -- caretakers that can take care of this pet type, with p_min
<= price <= p_max
   p_min DECIMAL(10, 2), -- note that if caretaker cannot take care of this pet type,
the price does not matter
   p_max DECIMAL(10, 2),
   p_start_date date, -- caretakers that can work on this interval
   p_end_date date
) RETURNS table (
   email varchar,
   name varchar,
   rating DECIMAL(10, 2),
   is_fulltime boolean,
   daily_price DECIMAL(10, 2) -- this is null if no pet type is specified
)
language plpgsql
AS
$$
BEGIN
   if p_pet_type is null then
       return query
       select
           ECT.email,
           ECT.name,
           ECT.rating,
           ECT.is_fulltime,
           null::numeric as daily_price
       from (
           Caretakers CT NATURAL JOIN (
               select U1.email, U1.name from users U1
           ) U
       ) as ECT
       where
           (ECT.name LIKE ('%' || p_ct_name || '%') or p_ct_name is null) and
           (ECT.rating >= p_rating or p_rating is null) and
           (ECT.is_fulltime = p_is_fulltime or p_is_fulltime is null) and
           (p_start_date is null or p_end_date is null or canWork(ECT.email,
p_start_date, p_end_date));
   else
       return query
       select
           ECT.email,
           ECT.name,
           ECT.rating,
           ECT.is_fulltime,
           ECT.daily_price
       from (
           Caretakers CT NATURAL JOIN (
               select U1.email, U1.name from users U1
           ) U NATURAL JOIN (
               select * from takecareprice
           ) TCP
       ) as ECT
       where
           (ECT.name LIKE ('%' || p_ct_name || '%') or p_ct_name is null) and
           (ECT.rating >= p_rating or p_rating is null) and
           (ECT.is_fulltime = p_is_fulltime or p_is_fulltime is null) and
```

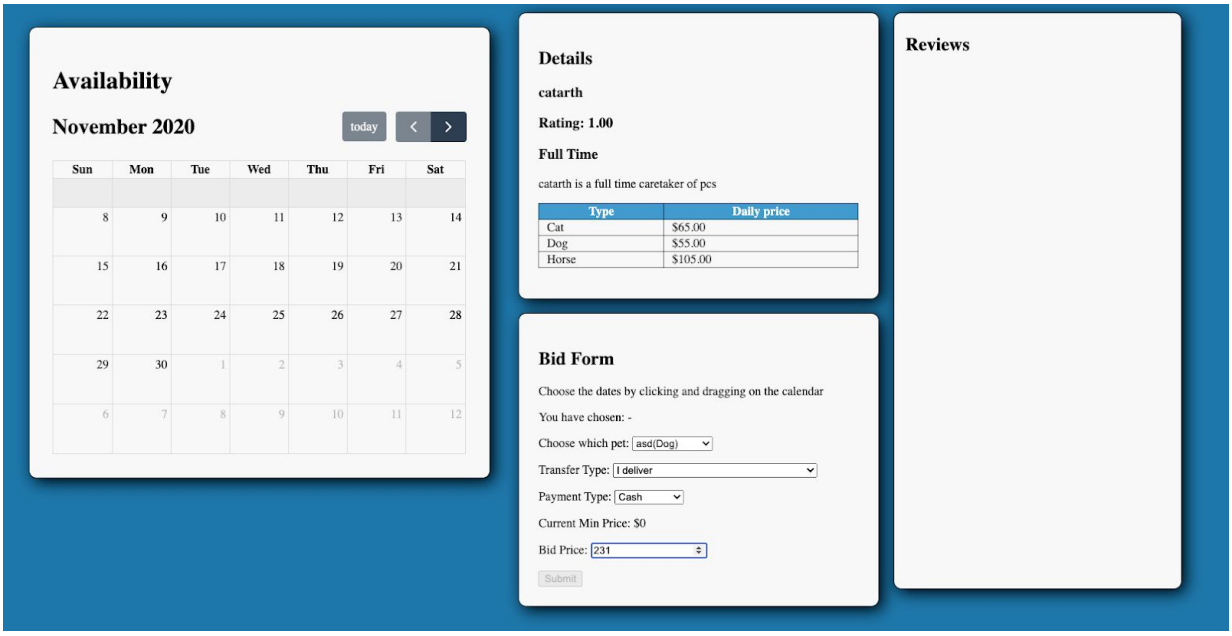
	(ECT.species = p_pet_type) and (ECT.daily_price >= p_min or p_min is null) and (ECT.daily_price <= p_max or p_max is null) and (p_start_date is null or p_end_date is null or canWork(ECT.email, p_start_date, p_end_date)); end if; END; $$;
PSQL query to compute daily demand and average demand on the interval '2020-01-01' to '2020-01-10' for pet type 'Dog'.	select COUNT(*) as num_days, SUM(demand) as total_demand, (SUM(demand) / COUNT(*))::DECIMAL(10, 2) as avg_demand from (select datez, (select COUNT(*) from bidsFor natural join (select email as owner_email, pet_name, species from pets) SP where start_date <= datez and datez <= end_date and species = 'Dog') as demand from (select generate_series('2020-01-01'::date, '2020-01-10'::date, '1 day'::interval)::date as datez) D) Dem
PSQL query to return salary to be paid to each caretaker for a month. This example is called for the month of Jan 2020, so 2020-01-01 to 2020-01-31	select email, name, CASE WHEN is_fulltime THEN 'Full Time' ELSE 'Part Time' END as type, description, rating, (case when (select is_fulltime from caretakers as i where i.email=o.email) and (case when (select sum(end_date - start_date + 1) from bidsfor where caretaker_email=o.email and ('2020-01-01' <= end_date and end_date <= '2020-01-31') and is_paid and is_confirmed group by caretaker_email) is null then 0 else (select sum(end_date - start_date + 1) from bidsfor where caretaker_email=o.email and ('2020-01-01' <= end_date and end_date <= '2020-01-31') and is_paid and is_confirmed group by caretaker_email) end) <= 60 then 3000 when (select is_fulltime from caretakers as i where i.email=o.email) and (case when (select sum(end_date - start_date + 1) from bidsfor where caretaker_email=o.email and ('2020-01-01' <= end_date and end_date <= '2020-01-31') and is_paid and is_confirmed group by caretaker_email) is null then 0 else (select sum(end_date - start_date + 1) from bidsfor where caretaker_email=o.email and ('2020-01-01' <= end_date and end_date <= '2020-01-31') and is_paid and is_confirmed group by caretaker_email) end) > 60

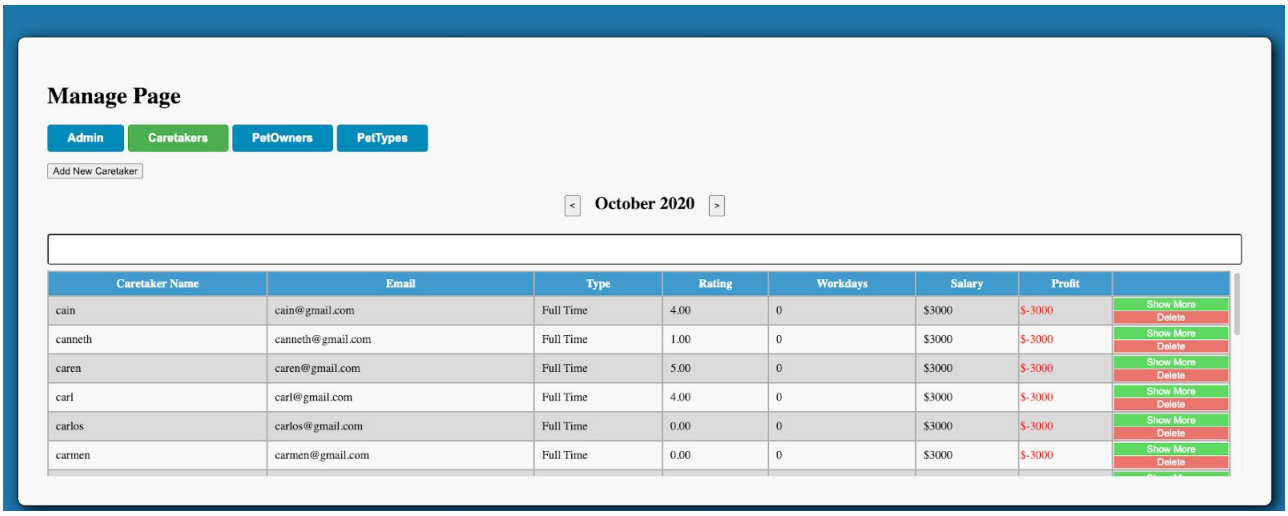
```
  then 3000 + (((case when (select sum(end_date - start_date + 1)
   from bidsfor
   where caretaker_email=o.email
    and ('2020-01-01' <= end_date and end_date <= '2020-01-31')
    and is_paid
    and is_confirmed
   group by caretaker_email) is null then 0 else (select sum(end_date - start_date +
1)
   from bidsfor
   where caretaker_email=o.email
    and ('2020-01-01' <= end_date and end_date <= '2020-01-31')
    and is_paid
    and is_confirmed
   group by caretaker_email) end) - 60) * ((case when (select sum((end_date -
start_date + 1) * amount_bidded)
    from bidsfor
    where is_paid
     and is_confirmed
     and ('2020-01-01' <= end_date and end_date <= '2020-01-31')
     and caretaker_email=o.email
    group by caretaker_email) is null then 0 else (select sum((end_date - start_date
+ 1) * amount_bidded)
    from bidsfor
    where is_paid
     and is_confirmed
     and ('2020-01-01' <= end_date and end_date <= '2020-01-31')
     and caretaker_email=o.email
    group by caretaker_email) end) / (case when (select sum(end_date - start_date +
1)
    from bidsfor
    where caretaker_email=o.email
     and ('2020-01-01' <= end_date and end_date <= '2020-01-31')
     and is_paid
     and is_confirmed
    group by caretaker_email) is null then 0 else (select sum(end_date - start_date +
1)
    from bidsfor
    where caretaker_email=o.email
     and ('2020-01-01' <= end_date and end_date <= '2020-01-31')
     and is_paid
     and is_confirmed
    group by caretaker_email) end)))
  else
  0.75 * (case when (select sum((end_date - start_date + 1) * amount_bidded)
   from bidsfor
   where is_paid
    and is_confirmed
    and ('2020-01-01' <= end_date and end_date <= '2020-01-31')
    and caretaker_email=o.email
   group by caretaker_email) is null then 0 else (select sum((end_date - start_date +
1) * amount_bidded)
   from bidsfor
   where is_paid
    and is_confirmed
    and ('2020-01-01' <= end_date and end_date <= '2020-01-31')
    and caretaker_email=o.email
   group by caretaker_email) end)
  end)::FLOAT as revenue
from users natural join caretakers as o
order by type asc, name asc;
```

9. Specification of software tools/frameworks used

Frontend was built using AngularJS, HTML and CSS. Backend was done with NodeJS and ExpressJS for its ease of use in creating REST API endpoints for our application. Databases and triggers were created with PostgreSQL and overall the application was hosted online using Heroku.

10. Representative Screenshots





11. Summary of difficulties encountered and lessons learnt

As most of us did not have any experience creating a full stack web application, there was a steep learning curve in learning about the stack used, including frontend and backend. There was also a wide range of functionality required to be implemented, which made it even more challenging as we were not familiar with the tech stack and had to implement complex functionalities at the same time. As there are separate responsibilities for frontend and backend, it was hard to communicate the requirements and resulted in much overhead and misunderstanding.

There was also some difficulty in trying to hit the required number of entity-relationship sets in the ER diagram as we had insufficient entity-relationship sets after the ER diagram review and were not sure of what additional features we could add to the application. We were also struggling with the problem deletion of entries like Users as mentioned before.

Over the course of the project, we found ourselves rewriting the backend sql many times. This happened for several reasons. Sometimes, when looking up a solution to a problem we faced, we would discover a new inbuilt PostgreSQL function that we could use, or discover a PostgreSQL syntax that was better suited to what we wanted to do. This helped us to improve the way we wrote our queries.
After we implemented a good portion of the project, we realized that there was a lot of duplicated code, so we refactored the duplicated parts into functions. This helped tremendously in improving the readability and maintainability of the code. It also made it much easier to detect and resolve bugs.

However, one area which could be improved is the code organization. Currently, the backend still feels somewhat disorganized, and not modular.