School of Computing
Semester 1 AY2020/2021

CS2102 – Database Systems

Pet Caring System

Team 39:

Qian Jie	A0185654U
Liao LiXin	A0185533B
Liu Kejun	A0185786H
Lim Tiang Jung Timothy	A0199669X
Pham Tran Tuan Linh	A0185491W

1. Project Responsibilities

Member	Responsible For
Qian Jie	Project management, Login/Register, Password hashing, user sessions, Implementing constraints, Computing salaries, delete, update
Li Xin	Pet owner/Caretaker homepage, Add pet, Add caretaker availability, Caretaker take leave
Kejun	Develop UI, Admin homepage, Add caretaker charges, Update the overall rating and charges for full-time caretaker after pet owner gives a rating, Implement constraints
Timothy	Caretaker to view all received bids, caretaker to accept bid, pet owner to confirm transaction
Tuan Linh	Develop UI, Sorting/Search function, Pet owner to search for suitable caretaker, Pet owner to bid for caretaker, Admin to browse data

2. Application's Data Requirements and Functionality

In our pet caring application, we provided a platform for all the pet lovers. No matter whether you are a pet lover that are looking for an extra income or you are a pet lover that are very busy who do not have time to take care of your pets and looking for someone else to do it for you, our pet caring application is here to provide a platform for all the pet lovers around the world.

In order for our application to work, the required data that we need are from the pet owner and the caretaker. Within each pet owner and caretaker, other related data will be collected to implement certain functionalities such as calculating the salary of the month for the caretakers that requires the amount of pet day that the caretaker has worked in the month and also all the charges that the caretaker charged for each services per pet day.

Our pet caring application is able to fulfill most of the basic requirements and functionalities mentioned in the project requirement such as :

a. Support the creation/deletion/update of data for the different users
b. Support data access for the different users
c. Support the browsing/searching of caretaker by pet owner
d. Support the browsing of summary information for PCS Administrator, caretaker and pet owner

Below is the additional functionality that we have implemented,

Hashed Password

Initially, when we were implementing the login/register system, we realised that the password for our registered user is stored directly into the user table in the database. We felt that it is not reasonable and safe for our application system. Hence, we decided to store the hashed password into the database instead.

The hashing algorithm that decided to use is known as bcrypt. It uses Blowfish to encrypt a magic string by using a key that is derived from the password. Later when a user enters a password, the key is derived again and if the ciphertext produced by encrypting with that key matches the stored ciphertext, the user is authenticated to log into the system.
As a result, the password produced by bcrypt is an irreversible and deterministic one which ensures the safety and privacy of our system for our user.

Application's Data Constraints

User

- Each user should only have a single password and NOT NULL
- Password must be hashed
- User name is NOT NULL
- Email is used as username and must be unique
- Pet owner and caretaker can share the same user account(with overlap)
- User can only be pet owner and caretaker (with covering)

Pet owner

- Each Pet owner must be uniquely identified by their email address
- Pet owner must own at least one pet

Caretaker (full time)

- Max 5 pets at a time
- Should not take care of pets that they cannot take care of
- Must work for a minimum of 2x150 consecutive days
- Cannot apply leave when there is at least one pet under their care

Caretaker (part time)

- Max 2 pets at a time (for low rating)
- Max 5 pets at a time (for good rating)
- Should not take care of pets that they cannot take care of

Pet

- A pet can only be taken care of by one care taker at a time.
- A pet can only be owned by one pet owner
- A pet is unique identified by pet ID
- Pet name is NOT NULL

PCS Admin

- Each admin should only has a single password and not null
- Each admin is identified by a unique Admin ID

Charge

- Charge amount cannot be negative integer or null

Bidsfor

- Bidding price cannot be negative integer or null
- Subsequent bidding amounts must be higher than the previous one by a minimum fixed integer amount.

TakenCareBy

- Start date must be before end date
- There are 2 transfers for each transaction
- There are only 3 transferring method
- Pet owner deliver
- Caretaker pick up
- Transfer through the physical building of PCS

Rate

- Rating can be from 1-5 stars (integer) and optional
- Rating comment can be optional (NULL)

Leave

- Start date must be before end date

PetCategory

- Pet category is uniquely identified by CategoryID

3. ER Model

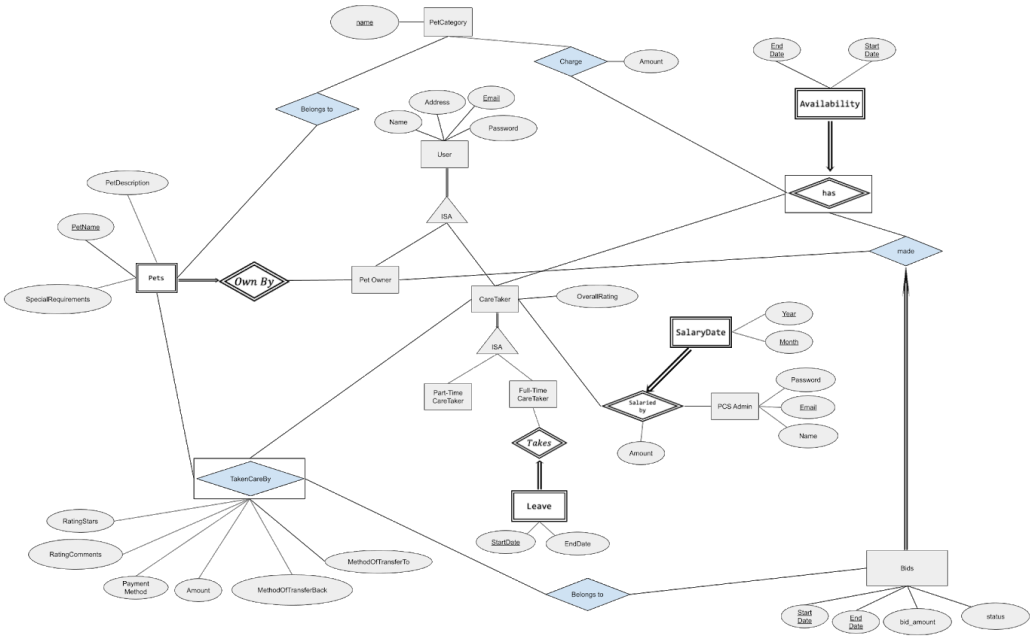

Figure 1: Our revised ER model

Constraints not captured by the ER diagram:

- Start date must before end date
- Full time caretaker can only take care of 5 pets at most at a time
- Part time caretaker can only take care of 5 pets at most at a time if with good rating and 2 pets at most at a time if with low rating
- A pet can only be taken care of by a caretaker at a time
- Full time caretaker must work for a minimum of 2x150 consecutive days
- Full time caretaker cannot apply leave when there is at least one pet under their care
- There is a base daily price for full-time caretakers and price can be higher depending on their ratings

4. Relational Schemas

```
CREATE TABLE IF NOT EXISTS pet_owner (
    email       VARCHAR PRIMARY  KEY,
    name        VARCHAR(64) NOT NULL,
    address     VARCHAR     NOT NULL,
    password    VARCHAR     NOT NULL
);

CREATE TABLE IF NOT EXISTS caretaker (
    email           VARCHAR PRIMARY  KEY,
    name            VARCHAR(64) NOT NULL,
    address         VARCHAR     NOT NULL,
    password        VARCHAR(64) NOT NULL,
    type            VARCHAR     NOT NULL,
    overall_rating  NUMERIC(3,2)
);

CREATE TABLE IF NOT EXISTS admin (
    email       VARCHAR PRIMARY  KEY,
    name        VARCHAR(64) NOT NULL,
    password    VARCHAR     NOT NULL
);

CREATE TABLE IF NOT EXISTS pet_category (
    category_name VARCHAR PRIMARY KEY,
    basic_charge  NUMERIC     NOT NULL
);

CREATE TABLE IF NOT EXISTS pets_own_by (
    pet_owner_email        VARCHAR REFERENCES pet_owner(email)
                                       ON DELETE CASCADE,
    pet_name               VARCHAR NOT NULL,
    special_requirements   VARCHAR,
    category_name VARCHAR REFERENCES pet_category(category_name),
    PRIMARY KEY (pet_owner_email, pet_name),
    UNIQUE (pet_name)
);

CREATE TABLE IF NOT EXISTS caretaker_has_availability (
      caretaker_email VARCHAR REFERENCES caretaker(email),
      start_date DATE,
      end_date DATE,
      PRIMARY KEY (caretaker_email, start_date, end_date),
      CHECK(start_date <= end_date)
);
```

```
CREATE TABLE IF NOT EXISTS caretaker_has_charge (
    caretaker_email  VARCHAR REFERENCES caretaker(email),
    category_name VARCHAR REFERENCES pet_category(category_name),
    amount NUMERIC,
    PRIMARY KEY (caretaker_email, category_name)
);

CREATE TABLE IF NOT EXISTS pet_owner_bids_for (
    pet_owner_email VARCHAR REFERENCES pet_owner(email),
    caretaker_email VARCHAR REFERENCES caretaker(email),
    pet_name        VARCHAR REFERENCES pets_own_by(pet_name),
    startdate       DATE NOT NULL,
    enddate         DATE NOT NULL,
    amount          NUMERIC NOT NULL,
    status          BIT DEFAULT 0::BIT NOT NULL,
    PRIMARY KEY (pet_owner_email, caretaker_email,
                 pet_name, startdate, enddate),
    CHECK(startdate <= enddate)
);

CREATE TABLE IF NOT EXISTS pets_taken_care_by (
      pet_owner_email VARCHAR REFERENCES pet_owner(email)
                                      ON DELETE CASCADE,
      pet_name VARCHAR REFERENCES pets_own_by(pet_name),
      caretaker_email VARCHAR REFERENCES caretaker(email),
      start_date DATE NOT NULL REFERENCES pet_owner_bids_for(startdate),
      end_date DATE NOT NULL REFERENCES pet_owner_bids_for(enddate),
      payment_method VARCHAR(64) NOT NULL,
      amount NUMERIC NOT NULL,
      method_to VARCHAR(64) NOT NULL,
      method_from VARCHAR(64) NOT NULL,
      rating_stars NUMERIC,
      rating_comment VARCHAR,
      PRIMARY KEY(pet_owner_email, pet_name,
                  caretaker_email, start_date)
);

CREATE TABLE full_time_takes_leave(
      caretaker_email  VARCHAR REFERENCES caretaker(email)
                                      ON DELETE CASCADE,
      start_date DATE NOT NULL,
      end_date DATE NOT NULL,
      PRIMARY KEY(caretaker_email, start_date),
      CHECK(start_date <= end_date)
);
```

```
CREATE TABLE caretaker_salaried_by(
      caretaker_email  VARCHAR REFERENCES caretaker(email),
      admin_email VARCHAR REFERENCES admin(email),
      amount NUMERIC NOT NULL,
      year INTEGER NOT NULL,
      month INTEGER NOT NULL,
      PRIMARY KEY(caretaker_email, year, month)
);
```

5. Discussion on 3NF or BCNF

No, our database is not on 3NF.
Counter example: table pets_taken_care_by
There exists at least a transitive functional dependency. For example, changing the non-key column rating_stars may change rating_comment.

6. Triggers

Trigger #1

This is a trigger used to address the constraint where the full-time caretaker must work for a minimum 2 x 150 days consecutively. This also means that whenever they are declaring their available days, they have to specify a range of days more than or equal to 150 days.

Hence, in figure 2, the function caretaker_availability() will be triggered whenever a full-time caretaker is selecting his/her available days. After it is triggered, the function will first check whether the caretaker falls under the full-time category and then check whether his/her input range of available date are more than or equal to 150 days, if yes then an insertion of new rows with his/her available days will processed, else the insertion process will be aborted.

```
CREATE OR REPLACE FUNCTION
caretaker_availability() RETURNS TRIGGER AS
$$ DECLARE ct NUMERIC;
    BEGIN
        SELECT COUNT(*) INTO ct FROM caretaker C
        WHERE NEW.caretaker_email = C.email AND C.type = 'full_time';
        IF ct > 0 AND NEW.end_date - NEW.start_date < 150 THEN
            RETURN NULL;
        ELSE
            RETURN NEW;
        END IF;
        END; $$
LANGUAGE plpgsql;

CREATE TRIGGER
check_fulltime_availability
BEFORE INSERT OR UPDATE ON caretaker_has_availability
FOR EACH ROW EXECUTE PROCEDURE caretaker_availability();
```

Figure 2: Code snippet of the implementation of Trigger #1

Trigger #2

This is the trigger used to address the constraint where a pet can only be taken care of by one caretaker at a time. This trigger will be activated whenever a pet owner is trying to bid a caretaker for a pet. Before inserting the bid into pet_owners_bids_for table which is the table used to store bid data, the trigger function will retrieve the data row from the pets_taken_care_by, which is the table used to store the transaction data by checking the condition that the start date and end date of the new bid overlaps with the start date and end date of the all transactions belong to this pet. If the data row retrieved from the database is greater than, which means there is already a transaction overlapping with the new bid, then the trigger function will return NULL and not insert the bid data into the database, otherwise, the new bid will be inserted into the database.

```
CREATE OR REPLACE FUNCTION
being_taken_care() RETURNS TRIGGER AS
$$ DECLARE ct NUMERIC;
    BEGIN
        SELECT COUNT(*) INTO ct FROM pets_taken_care_by ptcb
        WHERE NEW.pet_owner_email = ptcb.pet_owner_email AND NEW.pet_name = ptcb.pet_name
        AND NEW.startdate >= ptcb.start_date AND NEW.enddate <= ptcb.end_date;
        IF ct > 0 THEN
            RETURN NULL;
        ELSE
            RETURN NEW;
        END IF;
        END; $$
LANGUAGE plpgsql;

CREATE TRIGGER
check_being_taken_care
BEFORE INSERT OR UPDATE ON pet_owner_bids_for
FOR EACH ROW EXECUTE PROCEDURE being_taken_care();
```

Figure 3: Code snippet of the implementation of Trigger #2

Trigger #3

This trigger is used to address the constraint where one caretaker cannot take care of more than 5 pets at the same time. The function pet_limit() will be triggered when a caretaker is trying to accept a bid offered by a pet owner. When a pet owner accepts a bid, the status of this bid will be updated. To ensure that the caretaker cannot accept more than 5 bids at the same time, we count the number of records belonged to the caretaker which have the overlapped start date and end date with the bidding start date and the bidding end date in the table pets_owner_bids_for. There are overlapping dates if any of the two conditions is satisfied. These two conditions include if the bidding start date is before or the same as the start date of a record and the bidding end date is after the start date of the record, or if the bidding start date is after the start date of a record and the bidding start date is before or same as the end date of the record.

```
CREATE OR REPLACE FUNCTION
pet_limit() RETURNS TRIGGER AS
$$ DECLARE ct NUMERIC;
    BEGIN
        SELECT COUNT(*) INTO ct FROM pets_owner_bids_for pobf
        WHERE NEW.caretaker_email = pobf.caretaker_email AND pobf.status = 1 AND NEW.status = 1
        AND ((NEW.startdate <= pobf.startdate and NEW.enddate >= pobf.startdate) or
        (NEW.startdate >= pobf.startDate and NEW.startdate <= pobf.endDate));
        IF ct >= 5 THEN
            RETURN NULL;
        ELSE
            RETURN NEW;
        END IF;
    END; $$
LANGUAGE plpgsql;

CREATE TRIGGER
check_pet_limit
BEFORE UPDATE ON pet_owner_bids_for
FOR EACH ROW EXECUTE PROCEDURE pet_limit();
```

Figure 4: Code snippet of the implementation of Trigger #3

7. Complex SQL Queries

Query #1

This query is used when the pet owner searches for a suitable caretaker for one of his/her pets for a period of time. It will return a list of part-time caretaker able to take care of the selected pet during the time period specified.

```
SELECT * FROM
((caretaker c INNER JOIN caretaker_has_charge cc
ON c.email = cc.caretaker_email)
INNER JOIN caretaker_has_availability ca
ON c.email = ca.caretaker_email)
WHERE c.type = 'part_time'
AND cc.category_name = $1
AND ca.start_date <= $2
AND ca.end_date >= $3;
```

Query #2

This SQL query is a stored procedure used to handle the insert query for the caretaker_has_charge table, which is used to store the charge amount of each pet category for caretakers. Before inserting the charge data into the database, this SQL query first checks whether the caretaker is a full time or part time caretaker. For full time caretaker, instead of taking a charge input from the caretaker, the query will retrieve the basic charge amount that is predefined by the administrator and take it as the charge amount of a particular pet category. For part time caretaker, the query will insert the charge data taken from the user input into the database.

```
CREATE OR REPLACE PROCEDURE caretaker_charge
(input_email VARCHAR, input_category VARCHAR, input_charge NUMERIC) AS
$$ DECLARE
 caretaker_type VARCHAR;
 amount NUMERIC;
 BEGIN
   SELECT type FROM caretaker INTO caretaker_type WHERE email = input_email;
   SELECT basic_charge FROM pet_category INTO amount
   WHERE category_name = input_category;
   IF caretaker_type = 'part_time' THEN
     INSERT INTO caretaker_has_charge
     VALUES (input_email, input_category, input_charge);
   ELSE
     INSERT INTO caretaker_has_charge
     VALUES (input_email, input_category, amount);
   END IF;
 END; $$
language plpgsql;
```

Query #3

This SQL query is a stored procedure used to update the overall rating of a caretaker and update the charges of the caretaker if the caretaker is a full-time caretaker. When a pet owner gives a rating for a caretaker, this procedure will pull all the transactions belonging to the caretaker that have a rating and calculate the average rating. This average rating will be updated in the caretaker table as the overall rating. Since the overall rating affects the charges of a full-time caretaker, we will update the charges of this caretaker by pull the basic charge from the pet_category table and calculate the new charges where new charges = the original charges * (1 + average rating divided by 5).

```
CREATE OR REPLACE PROCEDURE caretaker_overall_rating(input_petowner_email
VARCHAR, input_pet_name VARCHAR, input_caretaker_email VARCHAR) as
$$ DECLARE
 averageRating NUMERIC;
 caretaker_type VARCHAR;
 category VARCHAR;
 originalCharge NUMERIC;
 begin
   SELECT type FROM caretaker INTO caretaker_type
   WHERE email = input_caretaker_email;
   SELECT AVG(rating_stars) FROM pets_taken_care_by INTO averageRating
   WHERE caretaker_email = input_caretaker_email;
   IF caretaker_type = 'full_time' THEN
      SELECT category_name FROM pets_own_by INTO category
      WHERE pet_owner_email = input_petowner_email
      AND pet_name = input_pet_name;
      SELECT basic_charge FROM pet_category INTO originalCharge
      WHERE category_name        = category;
      UPDATE caretaker_has_charge
      SET amount = originalCharge * (1 + averageRating/5)
      WHERE caretaker_email = input_caretaker_email
      AND category_name = category;
   END IF;
   UPDATE caretaker SET overall_rating = averageRating
   WHERE email = input_caretaker_email;
END; $$
language plpgsql;
```

8. Software Tools/Frameworks

bcrypt

This is a password-hashing function used in our application, where the password stored in the user's table will not be revealed for privacy and security reasons. Instead, only hashed value will be stored under the password column as shown in figure 5.

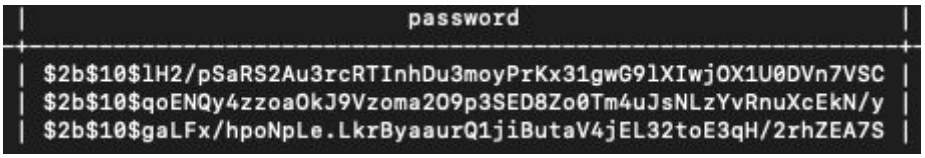

Figure 5: Hashed password stored in the user table

Passport.js

We used Passport.js for login authentication purposes. It supports various login types and callable functions for us to use during the configuration where we have to code the comparing logic behind the login authentication inside the passportConfig.js file.

By combining with bcrypt, we are able to compare the hashed password inside the login authentication.

express-flash

This is used for us to flash messages in the frontend views conveniently where the rendering of the page can be processed without redirecting the request.

express-session

This is an important module that enables the website to remember the users that have login and display the information or user state that is unique to this user.

node-postgres

This is a module that allowing us to connect and use all the functionalities that postgresSQL has provided such as querying data from table etc.

```
pool.query(`SELECT * FROM pets_own_by WHERE pet_owner_email = $1`, [email], (err, data) => {
  res.render('pet_owner_home', { title: 'Pet Owner Home', data: data.rows, user: req.user.name});
});
```

Figure 6: Code snippet showing the use of node-postgres

Async.js

This module is used mainly to prevent the asynchronous behaviour of SQL query. Detailed explanation and the reason behind that we choose to use this module has been discussed in section 10 (Lesson Learnt).

nodemon

This is a fantastic tool that greatly boosts our efficiency when working on the project. It is a tool that helps develop node.js based applications by automatically restarting the node application when file changes in the directory are detected. If there are any syntax errors, it will point it out to us and prevent us from making mistakes.

Express.js

This is probably the most important framework that we used for our project. It is a minimal and flexible web application framework designed for Node.js that provides a robust set of features for web

application and a wide range of APIs to be used to make our life easier in web development. One of the examples that we have used extensively is the “app.get()” and “app.post()” APIs.

EJS

This is a simple templating language that allows us to insert javascript code into HTML template. This is extremely useful when we want to display or hide information that renders on the web page by doing an if-else statement in the html file.

Bootstraps

This is a framework that we used for our frontend design and the overall looks of our web application. It helps us to design our website faster and easier. It also comes with responsive and mobile-first design.

9. Representative Screenshots of Our Application

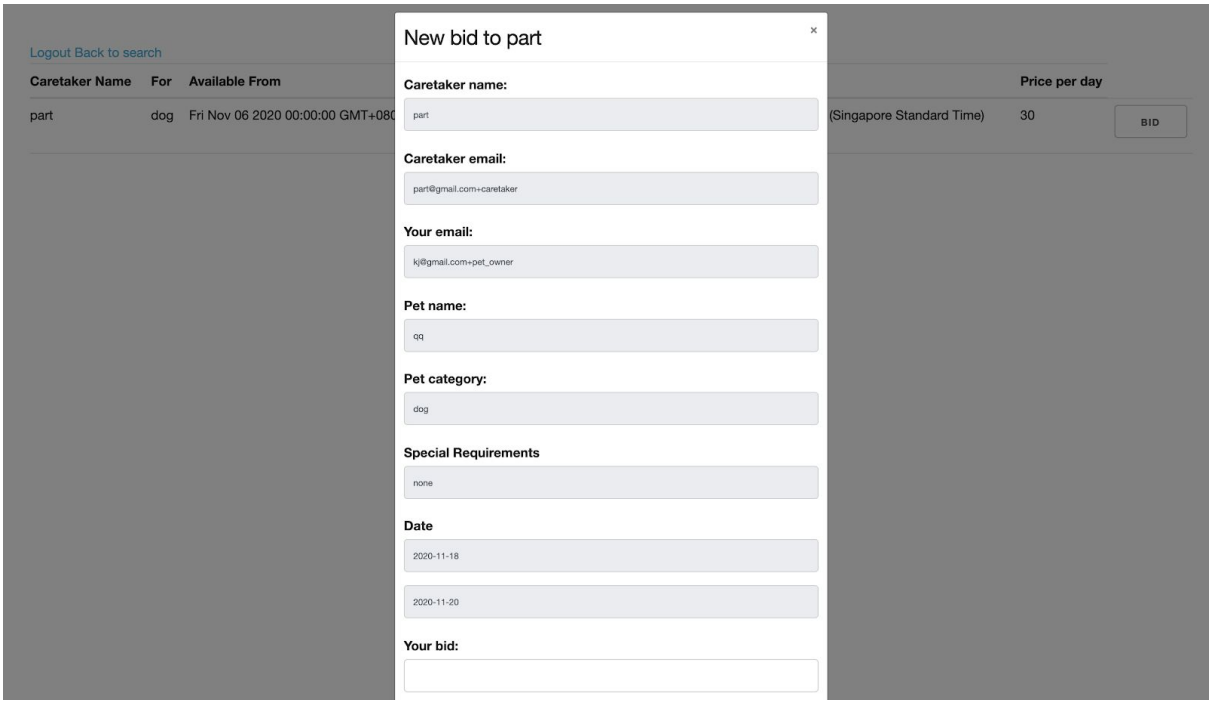

Figure 7: Representative Screenshots #1

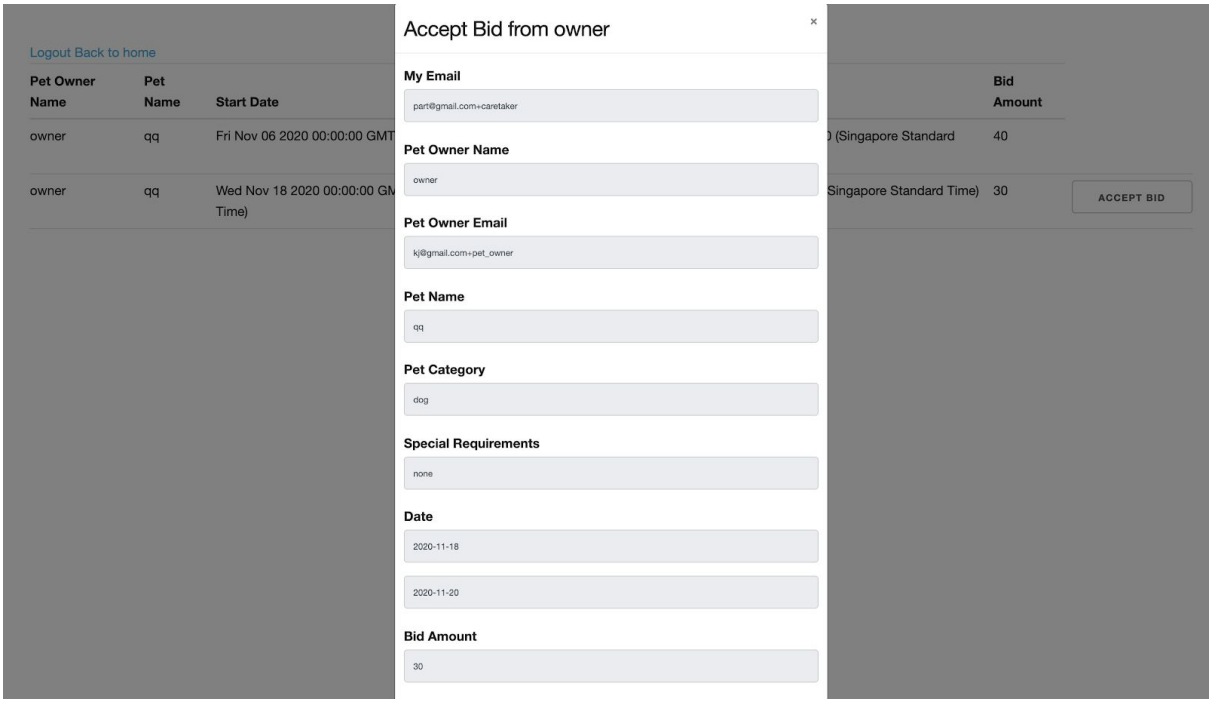

Figure 8: Representative Screenshots #2

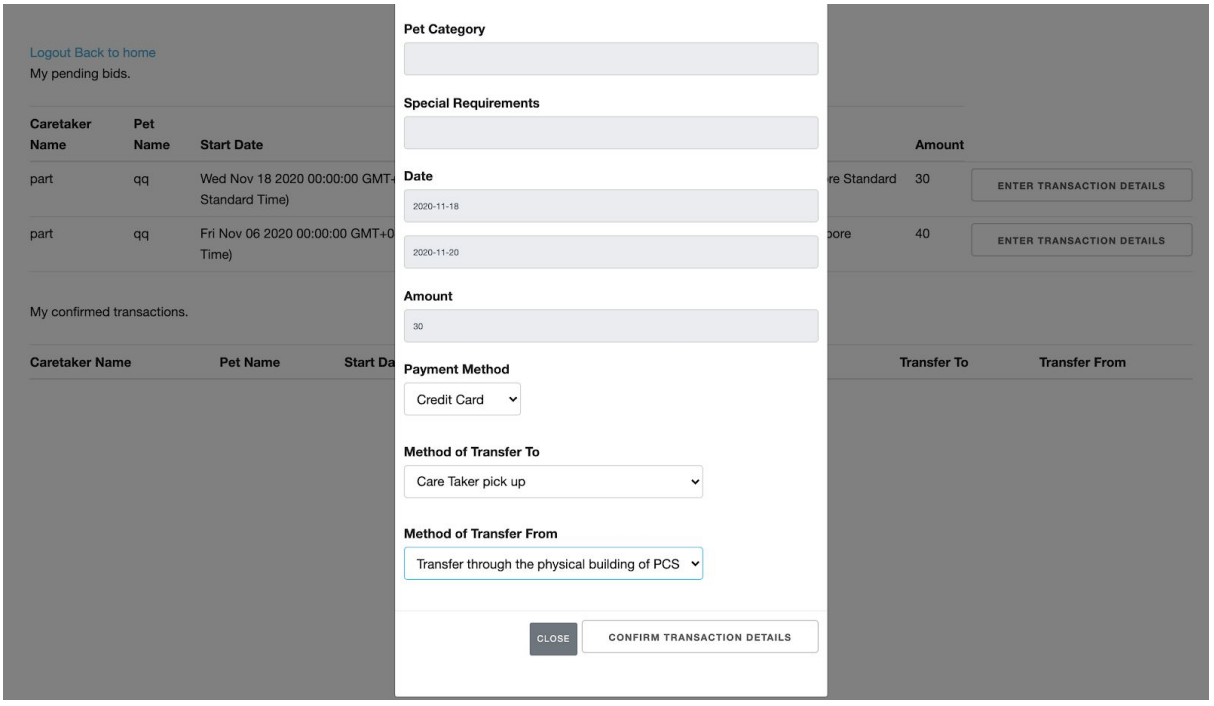

Figure 9: Representative Screenshots #3

10. Lesson Learnt

SQL query in a loop

When we are implementing the function of calculating the monthly salary of all the caretakers, we first need to query for all the caretaker from the caretaker's table and then using the primary key for each caretaker, we then query for the amount of days in a particular month that the caretaker has worked from pets_taken_care_by table to get the total pet day.

As you can see that this implementation requires a for loop to get each caretaker by query from the caretaker table and then another query is needed within the for loop to calculate the total pet day, as a result, we faced a problem where the output is incorrect as we realised that the caretaker for each iteration is not the caretaker that we are expecting it to be.

After a long time of debugging, we realised that it is due to the asynchronous nature of the SQL query. Hence, the SQL query will be computed asynchronously in a loop which means that these queries will not wait for the next iteration before they can be executed again. Hence, this messed up the caretaker for each iteration and thus the wrong output, since the caretaker in each iteration is always the caretaker in the next few iterations.

In order to resolve this issue, we make use of Async.js to prevent the asynchronous behaviour of the SQL query. By using the built in function "async.eachSeries()", we are able to query in a loop with synchronous behaviour. This ensures that the query is executed and waits for the loop to be completed before another query can be executed for the next iteration.

```
pool.query(
  `SELECT email, type FROM caretaker`,
  (err, results)=> {
    if (err) throw err;
    let caretakers = results.rows
    async.eachSeries(
      caretakers, function(caretaker, callback){
        pool.query(
          `SELECT amount, start_date, end_date FROM pets_taken_care_by ptcb
          WHERE ptcb.caretaker_email = $1 AND date_part('month', start_date) = $2
          AND date_part('year', start_date) = $3`,
          [caretaker.email, month, year],
          (err, results)=> {
            if (err) {
              callback(new Error('Failed to process' + caretaker));
            } else {
              callback(null);
            }
```

Figure 10: Code snippet of the use of async.eachSeries()

Using EJS to store SQL data accordingly

When the pet owner searches for a suitable caretaker by selecting his/her pet and the dates required, it will return a list of caretakers that are available and the pet owner can choose to bid for a suitable caretaker.

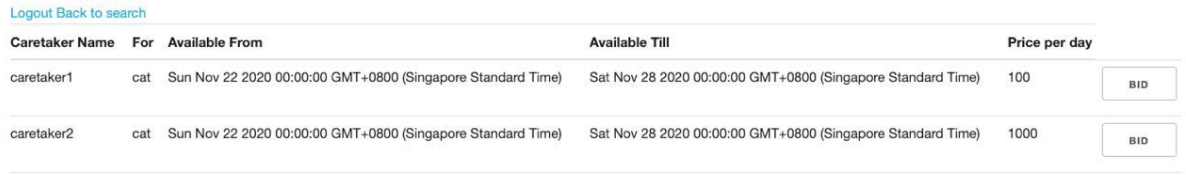

Figure 11: The list of caretakers that are able to take care of the selected part for that time period

At first, it was quite hard for us to implement the bid function as we would need to keep track of the caretaker that we are bidding for. Fortunately, we figured out that EJS is able to help us keep track of variables and call functions within HTML elements.

However, there are limited resources and documents about the use of EJS, especially for our case when we are calling functions using dynamic variables. Hence, only after countless times trying, we were able to implement that feature correctly.

What we did was to implement a Bootstrap 4 modal with the ability of varying modal content using event.relatedTarget and HTML data-* attributes to vary the contents of the modal depending on which button was clicked. The HTML data-* attributes are provided by the EJS variables depending on the row that we are on.

```
<td><button type="button" data-toggle="modal" data-target="#bidModal" data-start_date="<%= startDate%>"
  Bid</button></td>
```

Figure 12: Code snippet of the use of using EJS variables in HTML data- attributes*

```
$('#bidModal').on('show.bs.modal', function (event) {
var button = $(event.relatedTarget); // Button that triggered the modal
var caretaker_name = button.data('caretaker_name');
var caretaker_email = button.data('caretaker_email');
var petowner_email = button.data('petowner_email');
var pet_name = button.data('pet_name');
var pet_category = button.data('pet_category');
var pet_requirements = button.data('pet_requirements');
var start_date = button.data('start_date');
var end_date = button.data('end_date');
var modal = $(this);
modal.find('.modal-title').text('New bid to ' + caretaker_name);
$('#caretaker_name').val(caretaker_name)
$('#caretaker_email').val(caretaker_email)
$('#petowner_email').val(petowner_email)
$('#pet_name').val(pet_name)
$('#pet_category').val(pet_category)
$('#pet_requirements').val(pet_requirements)
$('#start_date').val(start_date)
$('#end_date').val(end_date)

})
```

Figure 13: Code snippet of the use of accessing EJS variables in HTML data- attributes using jQuery*