CS2102 Project

Team 57

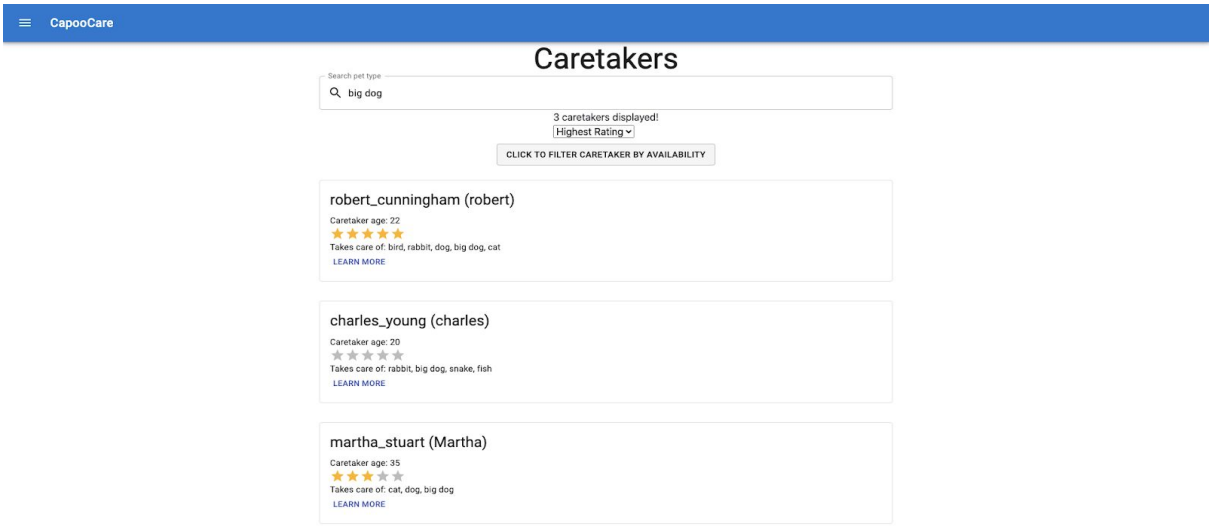
CapooCare

Matric No.	Name
A0183921A	Lim Yu Rong, Samuel
A0188200M	Jolyn Tan Si Qi
A0187652U	Nham Quoc Hung
A0190403R	Fyonn Oh Tong Shuang
A0189689W	Ng Qian Jie Cheryl

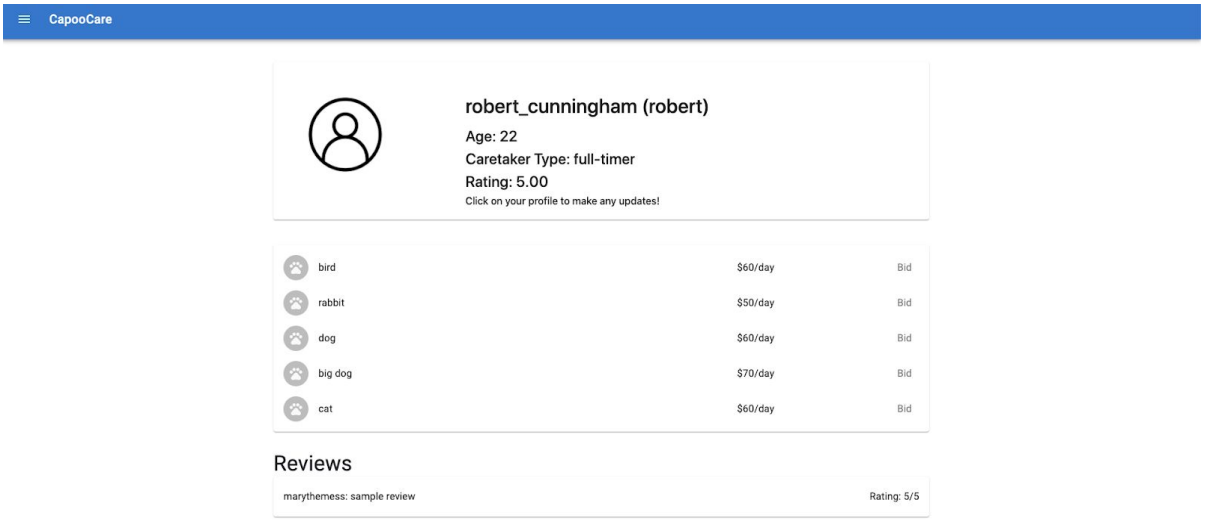
1. Introduction

1.1 Project Summary

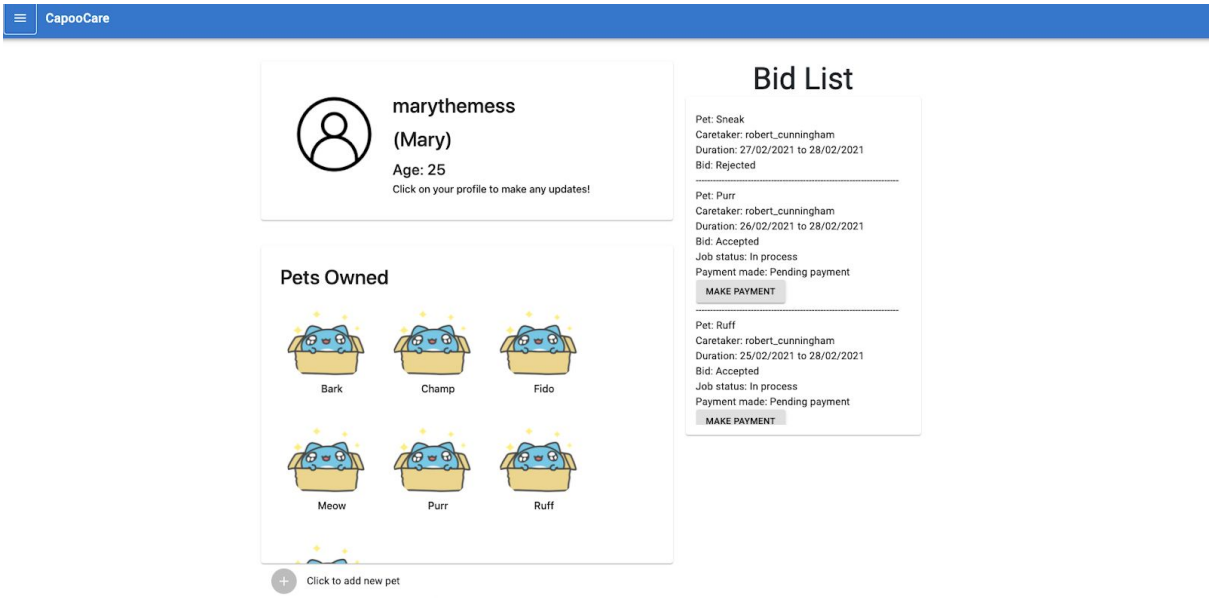
CapooCare is a platform that facilitates the exchange of pet care services between pet caretakers and pet owners. Pet owners can browse from a list of filtered caretakers according to their needs, as shown below.



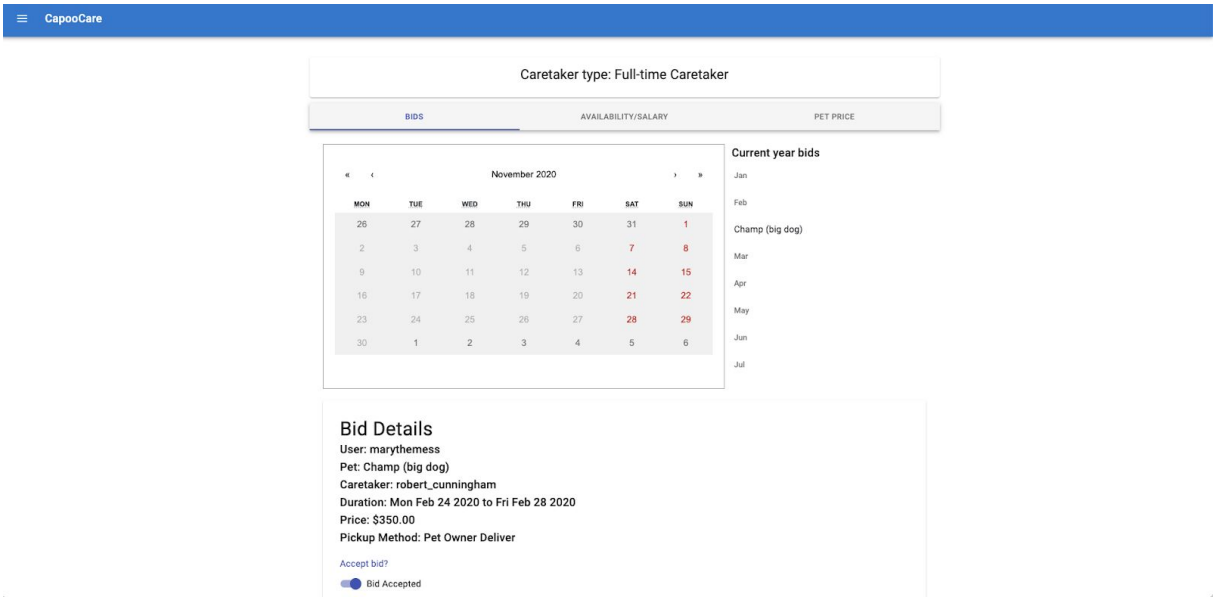
Pet owners are also able to preview the caretaker's profile, with information such as their rating and reviews before deciding to bid for a caretaker's service, as shown below.



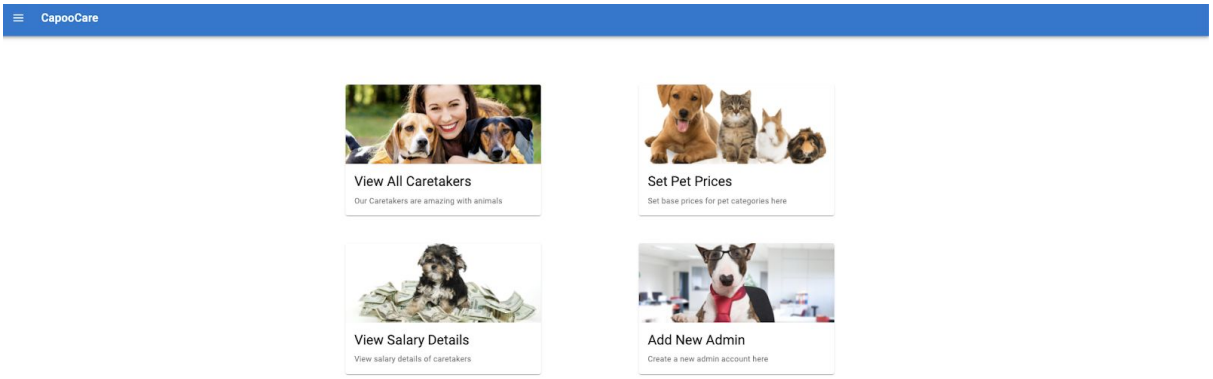
Pet owners can also view their previous bids and arrangements with caretakers, alongside with information on their own pets. Pet owners can also view pending bids, and leave reviews and ratings for completed bids, as can be seen in the screenshot below.



Users can also sign up to be caretakers, where they can take on bids by pet owners to care for pets for a period of time, depending on the pet types that they are comfortable with. Caretakers are able to sign up as Full-time or Part-time Caretakers, in which they would have different responsibilities. The diagram below shows details on the caretakers' bids according to dates displayed on the calendar.



CapooCare also supports administrative services for the admins, such as viewing the salary details of existing caretakers, as well as modifying the base prices of each pet category for the caretakers. These can be accessed by the administrator via tabs as seen below.



1.2 Team Roles and Responsibilities

Jolyn	Creation of ER, constraints and analysis, frontend components, frontend to backend logic, some CRUD API creation.
Hung	Creation of ER, constraints and analysis, frontend components, frontend to backend logic, some CRUD API creation.
Samuel	Creation of ER, constraints and analysis, API creation, SQL implementation of schema, queries, triggers, and data mocking.
Fyonn	Creation of ER, constraints and analysis, API creation, SQL implementation of schema, queries, triggers, data mocking, and deployment to Heroku.
Cheryl	Some frontend components and logic.

1.3 Overview of Application

1.3.1 General functionality of application

CapooCare supports two types of accounts - *User* and *PCS Admin*. A *User* could either be a *Pet Owner*, or a *Caretaker*, or both. Each *Pet Owner* can own *Pets*, and create *Bids* for caretaking services provided by *Caretakers* over a specific date range. *Caretakers* can indicate *Categories* of pets that they can take care of at a stated daily price. The *PCS Admin* is able to handle administrative details such as viewing salary information of the *Caretakers* as well as setting base prices of *Categories* of *Pets*. The tables in 1.3.2 and 1.3.3 provide a more detailed description of the application's data constraints and functionalities.

1.3.2 Constraints and Functionalities of Entities

Entities	Constraints	Functionalities
User	● Must have a unique username ● Must have first name ● Must be either a caretaker or a pet owner, or both (i.e. overlap allowed, and covering constraint present)	● Able to sign up as a caretaker or pet owner after they've signed up as the other role (e.g. a User that is only a pet owner can sign up to also be a caretaker afterwards)
Pet Owner	● Must own at least one pet	● Able to add new pets and delete pets ● Able to edit pet age, type and requirements ● Able to search for caretakers to care for their pets according to availability and pet type, sorted by rating ● Pet Owners going for vacation can also look for a caretaker who is able to care for all of their pets according to pet type
Pet	● Must have: ○ A unique combination of owner's username, pet name and pet type ○ A pet age ● Must belong to exactly 1 pet type category	● Able to have special requirements indicated by the pet owner
Category	● The pets of pet owners must belong to one and only one of the existing categories in this table.	● PCS Admins are able to add new Categories with their base prices ● Users are unable to add new Categories
Caretaker	● Must be either a Full-timer or a Part-timer, and not both ● Must be able to care for at least one pet category ● Can only care for pets whose pet type is of a category that	● Able to create new availability periods ● Able to indicate which categories of pet types they are able to care for ● Able to view their salary and

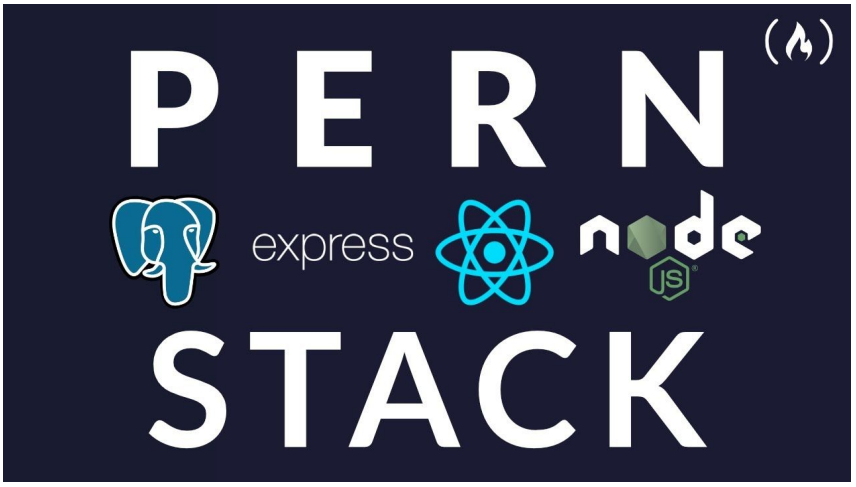
	they can take care of ● May take care of more than one pet at any given time, subject to further requirements as mentioned below	pet-days for months in the current year ● Able to view lucrative pet categories that they are not already caring for
Full-timer	● Must work for 2 consecutive 150-day periods during a year ● Cannot care for more than 5 pets at once ● Unable to set their own daily prices for categories of pets, and must follow the base price set by the PCS Admin	● Able to create new availability periods of at least 150 days or more ● Unable to reject a bid by a Pet Owner if they are available and are caring for less than 5 pets
Part-timer	● Can care for at most 2 pets at the same time if their rating is below 4 out of 5 ● Can care for at most 5 pets at the same time if their rating is at least 4 out of 5	● Able to specify their availability of any length within the next two years ● Able to set their own daily rates for categories of pet types
PCS Administrator	● Cannot be a User ● Must pay a full-time Caretaker a base salary of $3000 for up to 60 pet-days, with 80% bonus for other pet-days in that month ● Must pay a part-time Caretaker 75% of the price of pet-days	● Must be able to set base daily prices for Categories ● Must be able to view summary information pertaining to caretakers' salaries and pet days

1.3.3 Constraints on Relations between Entities

Relations	Constraints
Bid	**Creating bids** ● A user who is both a pet owner and a caretaker cannot bid for his or her own caretaking services ● A bid must be accepted by the caretaker before it can proceed (this process could be automatic for the fulltimer) ● A pet owner cannot bid for services for a pet during a timeframe where the pet already has a designated caretaker ● A pet owner can make multiple bids for one pet for overlapping/similar periods as long as none of the bids were accepted. Once one of the bids are accepted, the other pending bids will be rejected. ● A pet owner can only bid for a timeframe where the caretaker is available ● A pet owner with a specific pet of a certain pet type can only bid for a caretaker who is able to care for that pet type ● A pet owner cannot bid for a caretaker's availability that was in the past. **Payment, review and rating** ● A pet owner can make payment anytime after the bid was accepted ● A pet owner can leave an integer rating of 0 to 5 for the caretaker, along with a short review ● A pet owner can only leave a rating and review after the job has been completed and payment has been made

Has Availability	• Availability periods for a single Caretaker cannot overlap • Availability can only be specified for the next two years • Availabilities for Fulltimers must be 150 days long at least
Cares	• The constraints for these relationships have been sufficiently described above under Pet Owner, Pets and Caretakers.
Belongs	
Owns	

1.4 Project Stack

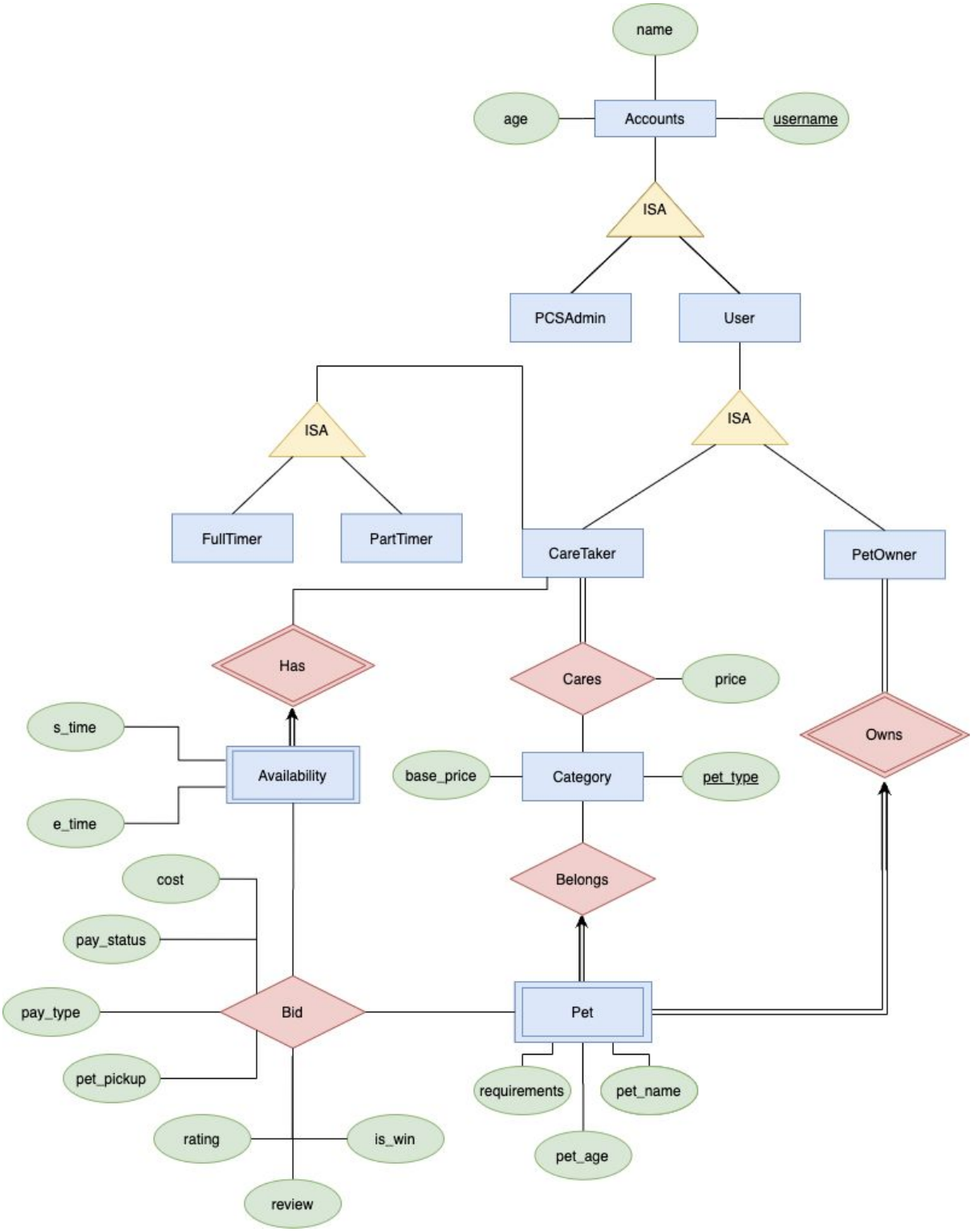


Our project was built on the PERN stack, with PostgreSQL as the database system, Express.js as the back-end framework, React as the frontend library and Node.js as the runtime environment. Additionally, we made use of Easy-Peasy, an abstraction of Redux, to maintain frontend states.

2. Database Model

2.1 ER Model

2.1.1 ER Diagram



2.1.2 Constraints not modelled by ER diagram

This section will discuss constraints of ***CapooCare*** that are not modelled in the ER diagram.

- Caretakers may take care of more than one pet at any given time.
- Full-time caretakers must work for 2 consecutive 150-day periods for each year, the first two of which must be indicated at the point of signing up.
- Full-time caretakers cannot care for more than 5 pets at once.
- Full-time caretakers are unable to set their own daily prices for categories of pets, and must follow the base price set by the PCS Admin
- Part-time caretakers can care for at most 2 pets at the same time if their rating is below 4 out of 5.
- Part-time caretakers can care for at most 5 pets at the same time if their rating is at least 4 out of 5.
- A user who is both a pet owner and a caretaker cannot bid for his or her own caretaking services.
- A bid must be accepted by the caretaker before it can proceed.
- A pet owner cannot successfully bid for services during a timeframe where the pet already has a designated caretaker.
- A pet owner can make multiple bids for one pet as long as none of the bids were accepted.
- A pet owner cannot bid for a caretaker's availability that was in the past.
- A pet owner with a specific pet of a certain pet type cannot successfully bid for a caretaker who is not able to care for that pet type.
- A pet owner can make payment any time after the bid was accepted.
- A pet owner can only leave a rating and review after the job has been completed and payment has been made.
- Availability periods for a single Caretaker cannot overlap.
- Availability can only be specified for the next two years.
- Other ISA constraints not captured by the ER diagram are mentioned in Section 2.1.3.

The functionalities of ***CapooCare*** have been adequately discussed in Section 1.1.3, and therefore will not be further elaborated upon here. Therefore, constraints that are implicitly functionalities (e.g. salary restrictions and definitions) will not be further discussed.

2.1.3 Non-trivial ER decisions

Note: *Pet* does not exhibit any weak entity set relationship with *Belongs*; it is instead a key and total participation constraint. The border around *Pet* refers to the relation *Owns* strictly.

- Only one attribute in this schema is defined like a serial type. This is the *username* attribute in the *Accounts* entity, which is indirectly inherited by *PCSAdmin*, *CareTaker*, and *PetOwner*.
    - The reasoning for this is that user accounts must be uniquely differentiated, and this is the most straightforward way of achieving this uniqueness. There is no other simple way of doing so; therefore this decision was made.
- There are several ISA constraints that are not reflected in the ER diagram:
    - All *Users* are a *PetOwner*, a *CareTaker*, or both (covering constraint, overlap constraint).
    - All *Accounts* are either a *PCSAdmin* or a *User* (covering constraint).
    - All *CareTakers* are either a *FullTimer* or a *PartTimer* (covering constraint).
- *Pet* has an identity dependency on *PetOwner*. In other words, *Pet* can only be uniquely identified in conjunction with the username of their *PetOwner*.
- *Availability* has an identity dependency on *CareTaker*. In other words, *Availability* can only be uniquely identified in conjunction with the username of its *CareTaker*.

2.2 Relational Schema

2.2.1 Entities

```
CREATE TABLE PCSAdmin (
   username         VARCHAR(50) PRIMARY KEY,
   adminName        VARCHAR(50) NOT NULL,
   age              INTEGER DEFAULT NULL
);

CREATE TABLE PetOwner (
   username         VARCHAR(50) PRIMARY KEY,
   ownerName        VARCHAR(50) NOT NULL,
   age              INTEGER DEFAULT NULL
);

CREATE TABLE CareTaker (
   username         VARCHAR(50) PRIMARY KEY,
   carerName        VARCHAR(50) NOT NULL,
   age              INTEGER DEFAULT NULL
);

CREATE TABLE FullTimer (
   username         VARCHAR(50) PRIMARY KEY REFERENCES CareTaker(username)
);

CREATE TABLE PartTimer (
   username         VARCHAR(50) PRIMARY KEY REFERENCES CareTaker(username)
);

CREATE TABLE Category (
   pettype          VARCHAR(20) PRIMARY KEY,
   base_price       INTEGER NOT NULL
);
```

2.2.2 Relations

```
CREATE TABLE Has_Availability (
   ctuname          VARCHAR(50) REFERENCES CareTaker(username) ON DELETE CASCADE,
   s_time           DATE NOT NULL,
   e_time           DATE NOT NULL,
   CHECK (e_time > s_time),
   PRIMARY KEY(ctuname, s_time, e_time)
);

CREATE TABLE Cares (
   ctuname          VARCHAR(50) REFERENCES CareTaker(username),
   pettype          VARCHAR(20) REFERENCES Category(pettype),
   price            INTEGER NOT NULL,
   PRIMARY KEY (ctuname, pettype)
```

```
);


CREATE TABLE Owned_Pet_Belongs (
   pouname          VARCHAR(50) NOT NULL REFERENCES PetOwner(username)
                                                    ON DELETE CASCADE,
   pettype          VARCHAR(20) NOT NULL REFERENCES Category(pettype),
   petname          VARCHAR(20) NOT NULL,
   petage           INTEGER NOT NULL,
   requirements     VARCHAR(50) DEFAULT NULL,
   PRIMARY KEY (pouname, petname, pettype)
);

CREATE TABLE Bid (
   pouname          VARCHAR(50),
   petname          VARCHAR(20),
   pettype          VARCHAR(20),
   ctuname          VARCHAR(50),
   s_time           DATE,
   e_time           DATE,
   cost             INTEGER,
   is_win           BOOLEAN DEFAULT NULL,
   rating           INTEGER CHECK((rating IS NULL)
                              OR (rating >= 0 AND rating <= 5)),
   review           VARCHAR(200),
   pay_type         VARCHAR(20) CHECK((pay_type IS NULL)
                         OR (pay_type = 'credit card') OR (pay_type = 'cash')),
   pay_status       BOOLEAN DEFAULT FALSE,
   pet_pickup       VARCHAR(20) CHECK((pet_pickup IS NULL)
                         OR pet_pickup = 'poDeliver' OR pet_pickup = 'ctPickup'
                         OR pet_pickup = 'transfer'),
   FOREIGN KEY (pouname, petname, pettype) REFERENCES
                              Owned_Pet_Belongs(pouname, petname, pettype),
   PRIMARY KEY (pouname, petname, pettype, ctuname, s_time, e_time),
   CHECK (pouname <> ctuname)
);
```

2.3 Database Normalization

- All tables in the database can be split into a primary key (containing some number of attributes), and some number of other attributes.
- By definition of primary key, the elements in the primary key are collectively unique, and can also uniquely identify all other attributes. All primary keys are keys, and therefore superkeys.
- No non-primary key attributes are able to uniquely identify any other attribute in the database, either individually or collectively. Therefore, no other attribute is part of any key, and therefore none of them are prime attributes.
- Therefore, all functional dependencies in each table are of the form ***a -> B*** where ***a*** comprises all attributes in the primary key, and ***B*** represents any single other attribute in the table.
- Therefore, all tables in the database are in BCNF. Since all tables in BCNF are also in 3NF, therefore all tables in the database are in 3NF as well.

2.4 Constraints Not Enforced by Schema

Constraint	**Enforcement**
A distinct pet owner and caretaker cannot have the same username e.g If a Pet Owner has already signed up with the username “marythemess”, a second user cannot sign up as a Caretaker or Pet Owner with the username “marythemess”.. However, if the user with username “marythemess” wants to be a Caretaker, he/she can do so within the application once she has signed in.	Enforced by frontend validation
A pet owner must own at least one pet.	Enforced by procedure ***add_petOwner***
A caretaker must either be a full-timer or part-timer, and must at least be able to care for one pet type belonging to a Category	Enforced with procedures ***add_fulltimer*** and ***add_parttimer***, and trigger ***not_parttimer_or_fulltimer***, ***not_parttimer*** and ***not_fulltimer***
Full-time caretakers must work for 2 consecutive 150 day periods during a year and are only able to add availability periods of 150 days or more, and both periods combined must be within 365 days	Enforced by the ***add_fulltimer*** procedure
Full-timer caretakers are unable to reject a bid by a Pet Owner if they are available and are caring for less than 5 pets	Enforced by the ***validate_mark***, ***mark_other_bids,*** and ***mark_bid_automatically_for_fulltimer*** triggers
Full-time caretakers are unable to set their own daily prices for categories of pets, and must follow the base price set by the PCS Admin	Enforced by the ***check_ft_care_price*** and ***check_update_base_price*** triggers
Part-time caretakers can care for at most two pets at the same time if their rating is below 4 out of 5 and at most 5 pets if their rating is at least 4 out of 5	Enforced by the ***validate_mark*** trigger
A pet owner cannot successfully bid for services during a timeframe where the pet already has a designated caretaker	Enforced by the ***validate_mark*** trigger and the ***add_bid*** procedure
A pet owner with a specific pet of a certain pet type can only bid for a caretaker who is able to care for that pet type, and can only bid for a timeframe where the caretaker is available	Enforced by the ***add_bid*** procedure
A pet owner can only leave a rating and review after the job has been completed and payment has been made	Enforced by the ***check_rating_update*** trigger
A pet owner cannot bid for a caretaker’s availability that was in the past.	Enforced via frontend validation

3. Triggers and Queries

3.1 Triggers

3.1.1 Validate Bid insertion or marking of Bid as successful

```
CREATE OR REPLACE FUNCTION validate_mark()
RETURNS TRIGGER AS
$$
DECLARE ctx NUMERIC;
DECLARE pet NUMERIC;
DECLARE matchtype NUMERIC;
DECLARE care NUMERIC;
DECLARE rate NUMERIC;
   BEGIN
       IF OLD.is_win = True THEN
           RETURN NEW;
       END IF;

       SELECT COUNT(*) INTO pet
           FROM Bid
           WHERE NEW.pouname = Bid.pouname AND NEW.petname = Bid.petname
             AND Bid.is_win = True
             AND (NEW.s_time, NEW.e_time) OVERLAPS (Bid.s_time, Bid.e_time);

       SELECT COUNT(*) INTO matchtype
           FROM Cares
           WHERE NEW.ctuname = Cares.ctuname AND NEW.pettype = Cares.pettype;

       IF pet > 0 THEN
           RAISE EXCEPTION 'Pet already cared for during that period.';
       ELSIF matchtype = 0 THEN
           RAISE EXCEPTION 'Caretaker cannot care for Pet type.';
       END IF;

       SELECT COUNT(*) INTO ctx
           FROM FullTimer F
           WHERE NEW.ctuname = F.username;
       SELECT COUNT(*) INTO care
           FROM Bid
           WHERE NEW.ctuname = Bid.ctuname AND Bid.is_win = True
             AND (NEW.s_time, NEW.e_time) OVERLAPS (Bid.s_time, Bid.e_time);

       IF ctx > 0 THEN -- If CT is a fulltimer
           IF care >= 5 AND NEW.is_win = True THEN
               RAISE EXCEPTION 'This caretaker has exceeded their capacity.';
           ELSE
               RETURN NEW;
           END IF;
```

```
        ELSE -- If CT is a parttimer
            SELECT AVG(rating) INTO rate
                FROM Bid AS B
                WHERE NEW.ctuname = B.ctuname;
            IF rate IS NULL OR rate < 4 THEN
                IF care >= 2 AND NEW.is_win = True THEN
                    RAISE EXCEPTION 'This caretaker has exceeded their capacity.';
                ELSE
                    RETURN NEW;
                END IF;
            ELSE
                IF care >= 5 AND NEW.is_win = True THEN
                    RAISE EXCEPTION 'This caretaker has exceeded their capacity.';
                ELSE
                    RETURN NEW;
                END IF;
            END IF;
        END IF;
    END; $$
LANGUAGE plpgsql;

CREATE TRIGGER validate_bid_marking
BEFORE INSERT OR UPDATE ON Bid
FOR EACH ROW
EXECUTE PROCEDURE validate_mark();
```

This trigger ensures that the marking of a Bid is valid. When a Bid is marked as won for a Caretaker, the conditions that must be fulfilled are:

- The Pet must not already be cared for by a Caretaker during that time frame
- The Caretaker must be able to care for the Pet's type
- The Caretaker must not have reached their limit of Pets cared for at that time

Firstly, this validation should only execute if the UPDATE that triggered this query specifically marked this query. If the query was previously already marked as won, then this trigger can simply RETURN NEW, without further validation.

Next, it will be checked whether the Pet already has a won Bid that overlaps this Bid. If it does, then it is impossible for both Bids to simultaneously be fulfilled, and therefore the validation for this Bid fails. This is implemented using the OVERLAPS clause, which checks whether two intervals of dates intersect each other.

Then, it will be checked using the Cares table whether the Caretaker involved is capable of caring for the Pet's type. If they cannot, then the validation for this Bid fails.

Finally, the number of Pets that the Caretaker will be caring for at that time will be calculated. If the Caretaker's existing number of Pets at that time frame exceeds their limit, the validation fails. This limit is 5 if the Caretaker is a Fulltimer, and either 2 or 5 if the Caretaker is a Parttimer, depending on their average rating.

If all checks pass, then the marking of the Bid is accepted. If any check fails, then an exception is raised and the UPDATE will fail.

This trigger will be checked whenever a new Bid is inserted into the Bid table, or whenever a Bid is marked as won. It will trigger before the insertion or update of a Bid.

3.1.2 Automatically mark a Fulltimer's Bid if possible

```
CREATE OR REPLACE FUNCTION mark_bid_automatically_for_fulltimer()
RETURNS TRIGGER AS
$$
DECLARE ft NUMERIC;
DECLARE bidcount NUMERIC;
   BEGIN
       SELECT COUNT(*) INTO ft
           FROM FullTimer F
           WHERE NEW.ctuname = F.username;
       SELECT COUNT(*) INTO bidcount
           FROM Bid
           WHERE NEW.ctuname = Bid.ctuname AND Bid.is_win = True
             AND (NEW.s_time, NEW.e_time) OVERLAPS (Bid.s_time, Bid.e_time);
       IF ft > 0 THEN -- If the Fulltimer has capacity
           IF bidcount < 5 THEN
               UPDATE Bid SET is_win = True
                    WHERE ctuname = NEW.ctuname AND pouname = NEW.pouname
                    AND petname = NEW.petname AND pettype = NEW.pettype
                    AND s_time = NEW.s_time AND e_time = NEW.e_time;
           ELSE
               UPDATE Bid SET is_win = False
                    WHERE ctuname = NEW.ctuname AND pouname = NEW.pouname
                    AND petname = NEW.petname AND pettype = NEW.pettype
                    AND s_time = NEW.s_time AND e_time = NEW.e_time;
           END IF;
       END IF;
       RETURN NEW;
   END; $$
LANGUAGE plpgsql;

CREATE TRIGGER fulltimer_automatic_mark_upon_insert
AFTER INSERT ON Bid
FOR EACH ROW
EXECUTE PROCEDURE mark_bid_automatically_for_fulltimer();
```

This trigger ensures that when a Bid is made for a Fulltimer, the Fulltimer will automatically accept the Bid if possible.

Firstly, it is checked that the Caretaker in the Bid is a Fulltimer. If the Caretaker is a Fulltimer, then the number of successful Bids that overlap with the inserted Bid are counted. This does not include the inserted Bid because it has not yet been marked as won.

If there are fewer than 5 successful overlapping Bids, then the inserted Bid is automatically marked as won. Otherwise, the trigger will mark this bid as lost (given that it cannot be won anymore).

3.1.3 Automatically updates all Fulltimer's rates with new base price

```
CREATE OR REPLACE FUNCTION check_update_base_price()
RETURNS TRIGGER AS
$$ BEGIN
        IF (NEW.base_price <> OLD.base_price) THEN
              UPDATE Cares SET price = New.base_price
                    WHERE (Cares.ctuname IN (SELECT username FROM FullTimer))
                    AND Cares.pettype = NEW.pettype;
        END IF;
        RETURN NEW;
    END; $$
LANGUAGE plpgsql;

CREATE TRIGGER check_update_base_price
BEFORE UPDATE ON Category
FOR EACH ROW EXECUTE PROCEDURE check_update_base_price();
```

This trigger ensures that whenever a category's base price is being updated by a PCSadmin, the prices for all FullTimers that can care for this category will change accordingly. This is done by updating the Cares table's rows that are from a FullTimer and of the specific category with the new updated price.

3.2 Complex Queries

3.2.1 Get Salary for all Parttimers

```
SELECT ctuname,
   SUM(cost) * 0.75 * (
       SELECT
           CASE
               WHEN AVG(rating) BETWEEN 4 AND 5
                   THEN 1.1
               WHEN AVG(rating) BETWEEN 3 AND 4
                   THEN 1.05
               ELSE 1
           END
           FROM Bid RIGHT JOIN Parttimer ON (Bid.ctuname = Parttimer.username)
           WHERE ctuname = username
   ) AS salary
   FROM (
       SELECT username AS ctuname, day, COALESCE(price, 0) AS cost,
                    pouname, petName
           FROM (
               SELECT
                   generate_series(
```

```
                    GREATEST(to_date($1, 'YYYYMMDD')::timestamp,
                                        s_time::timestamp),
                    LEAST(to_date($2, 'YYYYMMDD')::timestamp,
                                        e_time::timestamp),
                    '1 day'::interval
                ) AS day, price, ctuname, pouname, petName
                FROM Bid NATURAL JOIN Cares RIGHT JOIN Parttimer
                                    ON (Bid.ctuname = Parttimer.username)
                WHERE ctuname = username AND is_win = true
                    AND (s_time, e_time) OVERLAPS
                       (to_date($1, 'YYYYMMDD'), to_date($2, 'YYYYMMDD'))
                ORDER BY ctuname, day, price, pouname, petName
        ) AS totalprice RIGHT JOIN Parttimer
                            ON (totalprice.ctuname = Parttimer.username)
        GROUP BY username, day, price, pouname, petName
) AS salaries
GROUP BY ctuname;
```

This query calculates the salary of all Parttimers, and returns tuples of <ctuname, salary> for each ctuname that is a Parttimer.

The inputs used in this query are <s_time, e_time>, which indicate the start and end dates that are used in the querying for calculation of salary. For instance, if the salary for all Parttimer needs to be calculated for March 2021, then the input values are <'20210301', '20210331'>.

The workhorse of this query is the ***generate_series()*** function, which is used to generate all pet-days that match several conditions. These conditions are:

1. The Caretaker is a Parttimer (implemented using RIGHT JOIN on Parttimer)
2. The Caretaker has won the Bid (implemented via 'is_win = true')
3. The Bid timing must overlap the specified s_time and e_time (implemented via OVERLAPS)

Within the ***generate_series()*** function, the GREATEST and LEAST operators collectively limit the days examined to exactly within the confines of the input s_time and e_time, and the s_time and e_time of the Bid, whichever is a tighter bound. The *'1 day'::interval* option splits the timeframe into days.

This is then returned with the alias ***totalprice***, which represents all unique pet-days obtained by each Parttimer for the specified s_time and e_time, ordered and then grouped by ctuname, day, and price. A RIGHT JOIN on Parttimer is used to re-add the Parttimers that have not had any Bids for the timeframe, and the COALESCE operator is used to assign a price of 0 to these Parttimers (to indicate that they receive no bonuses). This is then returned with the alias ***salaries***. The SUM aggregate function then adds up all the bonuses obtained for each Parttimer. This is then multiplied by the 0.75 constant for bonuses.

Finally, an overall bonus of 5% or 10% is added to each salary depending on the average rating of the Parttimer. The CASE block handles the specific bonus value provided. Although the rating value of 4 satisfies both WHEN clauses, by order of the clauses, a rating of 4 will be assigned a bonus of 10%.

After each salary is calculated, it is then returned as a tuple with the ctuname of the respective Parttimer.

3.2.2 Get all available Caretakers that are able to care for all Petowner's pets within a period

```
SELECT DISTINCT
	A.ctuname,
	COALESCE(
		(SELECT AVG(rating)
			FROM Bid
			WHERE ctuname = A.ctuname
			GROUP BY ctuname)
	, 3) AS rating,
	(SELECT SUM(price) * (to_date($3,'YYYYMMDD') - to_date($2,'YYYYMMDD') + 1)
		AS days
		FROM Cares
		WHERE ctuname = A.ctuname AND pettype IN (
			SELECT DISTINCT pettype
				FROM Owned_Pet_Belongs
				WHERE pouname = $1
		)
	) AS price

	FROM  Has_Availability A
	WHERE NOT EXISTS (
		SELECT 1
			FROM (SELECT DISTINCT pettype
					FROM Owned_Pet_Belongs
					WHERE pouname = $1)
				AS PT
			WHERE NOT EXISTS (
				SELECT price
					FROM (SELECT DISTINCT pettype, price
						FROM Cares
						WHERE ctuname = A.ctuname) AS C2
					WHERE C2.pettype = PT.pettype
			)
	 )
	AND s_time <= to_date($2,'YYYYMMDD')
	AND e_time >= to_date($3,'YYYYMMDD')
	AND (
		(A.ctuname IN (SELECT username FROM Fulltimer)
			AND
			(SELECT COUNT(*)
				FROM Bid
				WHERE A.ctuname = Bid.ctuname AND Bid.is_win = True
				AND (to_date($2,'YYYYMMDD'), to_date($3,'YYYYMMDD'))
					OVERLAPS (Bid.s_time, Bid.e_time)
			) < 5
		) OR (
		   A.ctuname IN (SELECT username FROM Parttimer)
		   AND
		   CASE WHEN (SELECT AVG(rating)
```

```
                        FROM Bid AS B
                        WHERE  A.ctuname = B.ctuname) IS NULL
                    OR
                    (SELECT AVG(rating)
                        FROM Bid AS B
                        WHERE  A.ctuname = B.ctuname) < 4
                THEN (SELECT COUNT(*)
                        FROM Bid
                        WHERE A.ctuname = Bid.ctuname
                            AND Bid.is_win = True
                            AND (to_date($2,'YYYYMMDD'),
                                to_date($3,'YYYYMMDD'))
                            OVERLAPS (Bid.s_time, Bid.e_time)
                    ) < 2
            ELSE (SELECT COUNT(*)
                    FROM Bid
                    WHERE $1 = Bid.ctuname
                        AND Bid.is_win = True
                        AND (to_date($2,'YYYYMMDD'),
                            to_date($3,'YYYYMMDD'))
                            OVERLAPS (Bid.s_time, Bid.e_time)
                ) < 5
            END
        )
    ORDER BY rating DESC, price ASC;
```

This query obtains all available Caretakers that can take care of all the types of pets a specific Petowner has within a certain period. This query takes into account both the availability and the capacity of the caretaker to care for the pets at that period.

The inputs used in this query are <pouname, s_time, e_time>, which indicate the username of the Petowner to analyze, and the start and end dates which will be used as the period for the query. For example, if the Petowner “Mary” has pets of types: cat, rabbit and bird, and the period for querying is within “2020-11-01” to “2020-11-10”, the query will look for all caretakers (both full timers and part timers) that will be able to take care of type cat, rabbit and bird within the period “2020-11-01” to “2020-11-10”.

The workhorse of this query is the concept of Universal Quantification with the double negation effect, manifested via the “NOT EXISTS” operator. The effect results in a column of all Caretakers who are available to care for all types of pet the Petowners owns.

Within the ***WHERE*** clause, the query takes into account not just the availability of the Caretakers but also the capacity of them. This query ensures that if a caretaker has reached his/her limit of number of pets that can be taken care of at any given time (eg. Full timers can only take care of maximum 5 pets), they will be excluded from the results. This is done by checking if the total number of pets being taken care of within periods that overlap with the specified period has reached the limit.

The resultant table also includes the rating of the caretakers (generated by getting the average of ratings given to them in their past successful transactions) and the total cost of their services within the given period. The resultant table will then be ordered via their rating first in decreasing order, followed by their cost, in increasing order.

3.2.3 Get all Categories that a Caretaker cannot care for, ordered by lucrativeness

```
SELECT pettype, SUM(cost) AS lucrative_score
       FROM Bid
       WHERE pettype IN (
           SELECT pettype
               FROM Category
               WHERE pettype NOT IN (
                   SELECT pettype
                       FROM Cares
                       WHERE ctuname = $1
               )
       ) AND is_win = true AND s_time >= to_date($2, 'YYYYMMDD')
                           AND e_time <= to_date($3, 'YYYYMMDD')
       GROUP BY pettype
   UNION
   SELECT pettype, 0 AS lucrative_score
       FROM Category
       WHERE pettype NOT IN (
           SELECT pettype
               FROM Bid
               WHERE is_win = true
       ) AND pettype NOT IN (
           SELECT pettype
               FROM Cares
               WHERE ctuname = $1
       )
   ORDER BY lucrative_score DESC, pettype;
```

This query obtains all Categories (i.e. pet types) that a Caretaker cannot yet care for, and orders it in descending lucrativeness. A ‘lucrative’ pet type is one that has had a high amount of money involved in successful Bids for the specified timeframe.

The inputs used in this query are <ctuname, s_time, e_time>, which indicate the username of the Caretaker to analyze, and the start and end dates that are used in the querying for calculation of money flow. This could be used by Caretakers to identify which pet types they might want to train themselves to care for, given the demand for those pet types.

The ***lucrative_score*** of a pet type is the total amount of money involved in all successful Bids involving that pet type for a given timeframe. Two nested SELECT subqueries are used to obtain the list of all pet types that the Caretaker cannot yet care for. Then, the entries in Bid that match the pet type, are winning, and are entirely within the specified timeframe are selected, and their sum is returned as the lucrative_score.

Since this does not return pet types that have never been successfully bid for in the specified timeframe, these pet types must be added in through the UNION query. These represent all pet types that are not in any winning Bids, and also cannot be cared for by the Caretaker (remember that the point of this query was for the Caretaker to analyze the pet types that they are unable to care for). The combined result is finally ordered by descending value of lucrative_score.

4. Conclusion

4.1 Difficulties Faced

On the note of web development, it was hard to accurately deliver data starting from user input, into the Redux store, and then sending APIs to create queries in the database. Debugging was challenging and it was hard to identify the step where an error occurred. There were also a great deal of state management and design considerations when it came to the UI, especially with asynchronous API calls as well as the data constraints imposed by the database.

With regards to the database design, designing the ER diagram and thereafter translating that to SQL was challenging. The implementation of the constraints and the relevant details needed in the app required a lot of redesigning and refactoring. Debugging SQL code was very difficult as well because the error message was not very informative and this required a lot of time to solve. Additionally, due to us adding procedures and triggers in a somewhat *ad hoc* manner, it was usually hard to keep track of exactly where and when the validation for certain constraints was occurring. This likely led to us implementing multiple redundant layers of validation (e.g. once on the frontend, once in the procedure, and once again in the trigger). This was probably not best practice.

4.2 Lessons Learnt

We learnt that pair programming and consistent work are important in any project. Although many of us had previous experience with generating APIs, few of us had experience building a full-stack application and we needed to regularly consult each other on how to link our assigned sections together. We learnt that design decisions needed to be made early, and a feature freeze should have been implemented 1 week before the submission of the report. As it turned out, we were still modifying features up till the report submission date, and this led to a very hectic last few days.

We struggled with fine-tuning the schema and project requirements, especially given the tight time frame. However, with more experience and with a greater emphasis on planning in the future, we think that future projects will be smoother.