CS2102 Project Final Report (Team 27)

Team Members

e0253688: LIANG BOYUAN

e0253692: JIANG YIXING

e0424619: LI RUOCHEN

e0376962: LIU YANGMING

e0202053: YU JIAXIANG

Section I. Task Allocation

Ruochen is the coordinator of the project, and he takes charge of the framework setup and workflow designs. Yixing and Yangming take charge of the implementation of backend, including database queries. Jiaxiang and Boyuan take charge of the implementation of frontend.

All members are involved in the API design, database design and report writing.

Section II. Data requirements

Based on the project requirement, we have added the following additional features, constraints and interesting functionalities to our application:

Additional settings and data constraints:

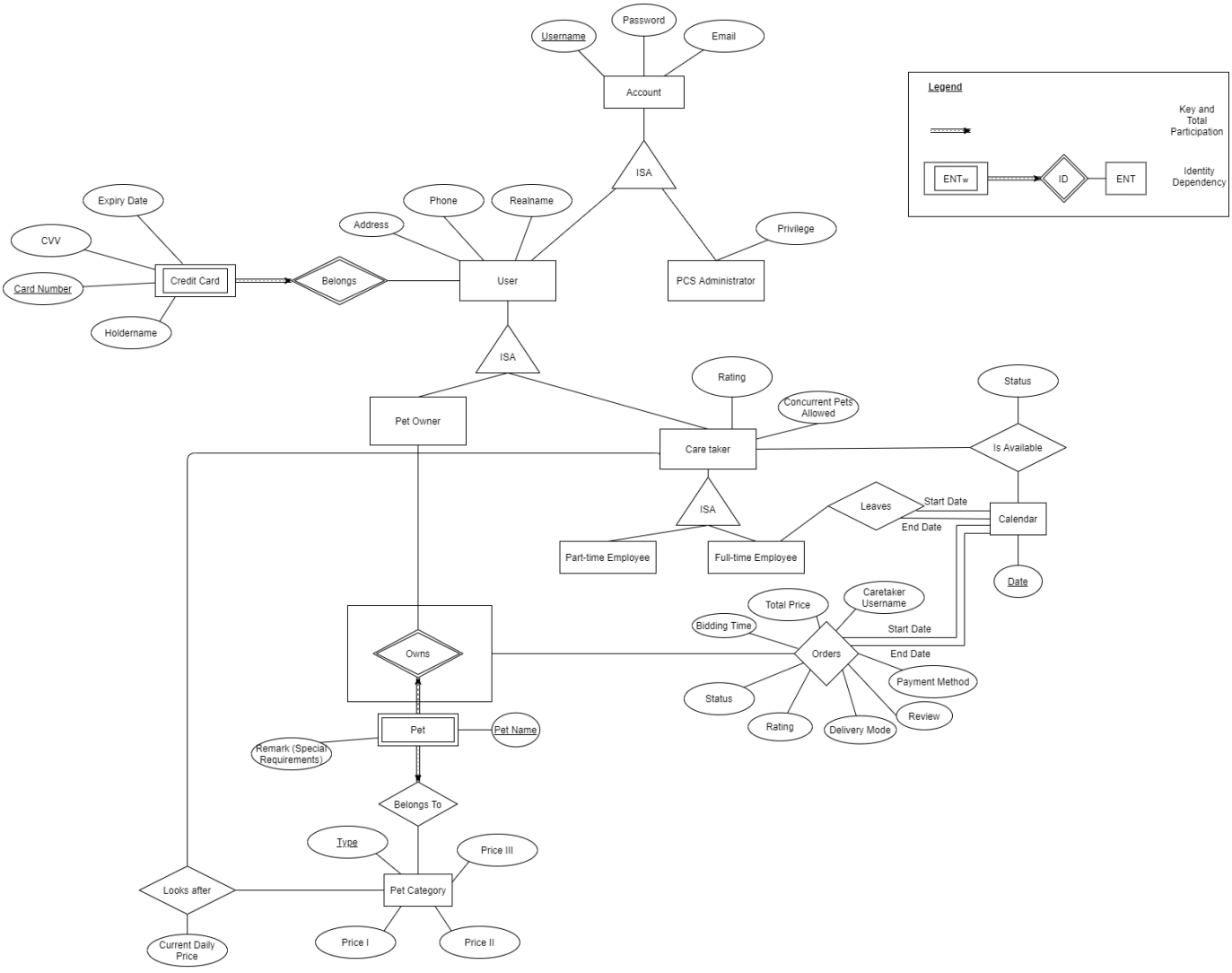
1. Each full-time caretaker can take care of any category of pet and is able to satisfy any additional remark. The daily price of full-time caretakers for each type of pet is specified by the PCS administrator.
2. The daily price of a full-time caretaker can vary based on his average rating. There are three bands of prices, 0-2 stars(price1), 2-4 stars(price2) and 4-5 stars (price3).
3. By default, a full-time caretaker is available all year around. If a full-time caretaker wants to make some specific days unavailable, he must apply for leave. The admin can view if the leave application conflicts with any existing orders (stored function), and then decide whether to approve the leave application or not.
4. A full-time caretaker cannot apply for leave on a day when there is a Pet to take care of (even when there is a pending payment order).
5. A full-time caretaker automatically accepts any orders from any petowners on his available slots.
6. Each part-time caretaker can specify the pet category he can take care of. The daily price of a part-time caretaker is specified by himself.
7. By default, a part-time caretaker is unavailable at any time. He can add available days by himself.
8. A part-time caretaker can choose to reject an order.
9. Each petowner does not have two pets with the same name.
10. Any user can register to become a part-time caretaker. To become a full-time caretaker, a user must be a part-time caretaker first, then apply to become a full-time caretaker. This will depend on the approval of the admin.
11. Once an order is submitted, the petowner cannot delete it.
12. Once a new pet is added, the petowner can no longer delete this pet or change the pet's name. But he can update the pet type and remark.
13. A full-time caretaker cannot apply for leave on days when he has orders, regardless of the order's payment status.

Interesting features:

1. For each available slot, the system automatically computes the average ratings of a caretaker and the total price. The petowner can also view all the past ratings and reviews of the specific caretaker.
2. The petowner can give ratings and reviews for each finished order (This means the order is accepted and the end date is before today). The petowner can update this rating even after a rating and review is already given.
3. The application automatically summarizes the number of pet-days (how many pets are taken care of in a given month for how many days) of a caretaker this month and the total value amount of orders this month.
4. The admin can see the total value amount of orders of all past months and the separated amount of each pet category.
5. When a user applies to be a part-time caretaker, he will be required to enter his real name if he has not done so. This is for the purpose of identification and convenience of pet owners when searching for a suitable caretaker.

Section III. ER diagram

Here is the overview of the ER diagram:



Non-trivial design decisions in ER diagram:

1. **IS-A:**
   a. An account is either a user or a PCS administrator, cannot be both (satisfies Covering constraint, but does not satisfy Overlap Constraint).
   b. A user is either a pet owner or a caretaker or both (satisfies Covering and Overlapping Constraint).
   c. A caretaker is either part-time or full-time, cannot be both (satisfies Covering constraint, but does not satisfy Overlap Constraint).

2. **Weak Entity:**
    a. Identity Dependency: Besides the credit card number, the identity of a credit card also depends on the user's username. This is to match the real-world situations where family members can share the same credit card.
    b. Identity Dependency: Besides pet name, the identity of a pet depends on the pet owner. This is to match the real-world situations where two pets can have the same pet name but belongs to different pet owners.
    c. Existential Dependency: The existence of a pet depends on the pet type, but a pet can be uniquely identified by its pet name and its pet owner's username. This is to match the real-world situations where pet caring facilities can only take in limited types of pets. Pets of types outside the specified pet types cannot be created.
3. **Aggregate:**
    a. Both petowner and petname will be part of the keys of owns, since it is possible for two different petowners to have pets with the same name.

The constraints not captured by ER diagram include:

1. A Pet Owner can only bid for service when the caretaker is available on every day during the service period.
2. A Pet Owner cannot submit duplicate bids for the same Pet under the same caretaker for the same period.
3. Each Account is either a User or PCS Administrator, cannot be both.
4. Each User is either a caretaker or Pet Owner or both (satisfies Covering and Overlapping Constraint).
5. Each caretaker is either a Full-time or Part-Time Employee (satisfies Covering constraint but does not satisfy Overlap Constraint).
6. For each Account, the Password and Email cannot be null. Email and Phone Number formats must be of valid style which is checked using RegEx.
7. For each Credit Card, CVV and Expiry Date cannot be null.
8. For Looks After, Daily Price cannot be null.
9. For Is Available, Status cannot be null.
10. For Orders, status, delivery mode and payment method cannot be null.
11. The Part time caretaker can only take care of Pets belonging to Pet Categories he/she can take care of.
12. A caretaker must not exceed the number of pets allowed to be taken care of at any given time.
13. The Pet Owner should specify only one of the three Delivery Modes given in the project description while bidding. We assume Full Time Care Takers accept any transfer method. Part Time caretaker shall consider the Delivery Mode while accepting the bid. Accepting the bid indicates accepting the Delivery Mode and guarantees availability.
14. For a Full-Time caretaker, we assume he can satisfy any special requirement due to training provided by PCS and take care of any kind of pet. Therefore, he will accept the order immediately if he can (available and does not exceed maximum number of pets). For a Part-Time caretaker, he can choose to accept or reject the order by himself.
15. The Pet Owner can give at most one Rating and Review after each completed caretaking service. The rating and review may be amended after given.

16. A Full-Time caretaker must have at least 2*150 consecutive available days each year.
17. A caretaker has one of the three status on each day, ‘available’ means on working and can take at least one more pet, ‘busy’ means on working but cannot take any additional pet. (Specifically, for Part Time Care Takers, it can be 2 or 5), ‘unavailable’ means not working/on leave. The status needs to be updated when an order status is changed.
18. An order can have a status of ‘Pending Caretaker Acceptance’, ‘Pending Payment’, ‘Payment Received’, or ‘Rejected Bid’.

Section IV. Relational schema

The following are the lists of relational schema used in our project:

```
CREATE TABLE accounts(
   username VARCHAR PRIMARY KEY,
   passwd VARCHAR NOT NULL,
   email VARCHAR NOT NULL UNIQUE,
   phone VARCHAR,
   addres VARCHAR,
   realname VARCHAR
);

CREATE TABLE admins(
   username VARCHAR PRIMARY KEY,
   email VARCHAR NOT NULL UNIQUE,
   passwd VARCHAR NOT NULL,
   privilege VARCHAR -- privilege for admins
);
```

```
CREATE TABLE cards(
   cardnumber VARCHAR(16),
   holdername VARCHAR NOT NULL,
   CVV VARCHAR(4) NOT NULL,
   expdate VARCHAR NOT NULL,
   username VARCHAR REFERENCES accounts(username),
   PRIMARY KEY (cardnumber, username)
);

CREATE TABLE pettypes(
   ptype VARCHAR PRIMARY KEY
);

CREATE TABLE pets(
   powner VARCHAR NOT NULL REFERENCES accounts(username) ON DELETE CASCADE,
   pname VARCHAR,
   remark VARCHAR,
   ptype VARCHAR REFERENCES pettypes(ptype) ON DELETE CASCADE,
   PRIMARY KEY(powner, pname)
);

CREATE TABLE caretakers(
   username VARCHAR PRIMARY KEY REFERENCES accounts(username),
   fulltime BOOLEAN DEFAULT FALSE,
   sumrating INT DEFAULT 0,
   numrating INT DEFAULT 0,
   maxpets INT, --maximum concurrent pets being taken
   apptime TIMESTAMP
);
```

```
CREATE TABLE looksafter(
   ctaker VARCHAR REFERENCES caretakers(username),
   price INT NOT NULL,
   ptype VARCHAR REFERENCES pettypes(ptype),
   PRIMARY KEY (ctaker, ptype)
);

CREATE TABLE fulltime_price(
   ptype VARCHAR PRIMARY KEY REFERENCES pettypes(ptype),
   price1 INT,
   price2 INT,
   price3 INT
);

CREATE TABLE calendar(
   date DATE PRIMARY KEY
);

CREATE TABLE available(
   ctaker VARCHAR REFERENCES caretakers(username),
   date DATE REFERENCES calendar(date),
   status VARCHAR DEFAULT 'available', --available, full
   PRIMARY KEY(ctaker, date)
);
```

```
CREATE TABLE leave(
    ctaker VARCHAR REFERENCES caretakers(username),
    startdate DATE REFERENCES calendar(date),
    enddate DATE REFERENCES calendar(date),
    clash VARCHAR DEFAULT 'false',
    status VARCHAR DEFAULT 'pending', --pending, approved
    PRIMARY KEY(ctaker, startdate, enddate)
);
CREATE TABLE orders(
    bidtime TIMESTAMP,
    powner VARCHAR NOT NULL,
    pname VARCHAR NOT NULL,
    ctaker VARCHAR,
    ptype VARCHAR NOT NULL,
    remark VARCHAR, --special requirement
    sdate DATE REFERENCES calendar(date),
    edate DATE REFERENCES calendar(date),
    CHECK(sdate <= edate),
    rating INT,
    price INT,
    delivery VARCHAR NOT NULL, --delivery mode
    payment VARCHAR NOT NULL, --payment method
    review VARCHAR,
    status VARCHAR,
    PRIMARY KEY (powner, pname, sdate, edate),
    FOREIGN KEY (powner, pname) REFERENCES pets(powner, pname),
    FOREIGN KEY (ctaker, ptype) REFERENCES looksafter(ctaker, ptype)
);
```

Section V. Normal Form

Most schemas in our implementations are in BCNF, but the following schemas are not in 3NF, so also not in BCNF:

1. Schema **looksafter**: There is redundancy in looksafter relation for full-time caretakers because (rating, ptype) -> daily price. This is not in BCNF, but it facilitates price computation of each order.

Section VI. Interesting triggers

1. **Automatically accept bid for full time caretakers**

```
CREATE OR REPLACE FUNCTION accept_bid()
RETURNS TRIGGER AS
$$ DECLARE ft BOOLEAN;
BEGIN
SELECT fulltime INTO ft FROM caretakers WHERE username = NEW.ctaker;
IF ft = true THEN
    NEW.status = 'Pending Payment';
END IF;
RETURN NEW;
END; $$
LANGUAGE plpgsql;

CREATE TRIGGER accept_bid_fulltime
BEFORE INSERT ON orders
FOR EACH ROW EXECUTE FUNCTION accept_bid();
```

2. Update caretaker rating & maximum pets allowed when a pet owner submits rating

```
CREATE OR REPLACE FUNCTION update_rating()
RETURNS TRIGGER AS
$$
DECLARE ft BOOLEAN;
BEGIN
SELECT fulltime INTO ft FROM caretakers WHERE username = NEW.ctaker;
IF OLD.rating IS NULL THEN
    UPDATE caretakers SET numrating = numrating + 1 WHERE username = NEW.ctaker;
    UPDATE caretakers SET sumrating = sumrating + NEW.rating WHERE username = NEW.ctaker;
ELSE
    UPDATE caretakers SET sumrating = sumrating + NEW.rating - OLD.rating WHERE username = NEW.ctaker;
END IF;
CALL update_maxpets(NEW.ctaker);
IF fulltime = true THEN
    UPDATE looksafter L SET price = (SELECT price1 FROM fulltime_price F WHERE F.ptype=L.ptype) WHERE ctaker=NEW.ctaker AND get_rating(ctaker)<=2;
    UPDATE looksafter L SET price = (SELECT price2 FROM fulltime_price F WHERE F.ptype=L.ptype) WHERE ctaker=NEW.ctaker AND get_rating(ctaker)>2 AND get_rating(ctaker)<=4;
    UPDATE looksafter L SET price = (SELECT price3 FROM fulltime_price F WHERE F.ptype=L.ptype) WHERE ctaker=NEW.ctaker AND get_rating(ctaker)>4;
END IF;

RETURN NEW;
END; $$
LANGUAGE plpgsql;


CREATE TRIGGER update_rating
AFTER UPDATE OF rating ON orders
FOR EACH ROW EXECUTE FUNCTION update_rating();
```

3. Update available relation when new orders are confirmed

```
CREATE OR REPLACE FUNCTION update_available()
RETURNS TRIGGER AS
$$ DECLARE maxpet INTEGER;
BEGIN
SELECT maxpets INTO maxpet FROM caretakers WHERE username = NEW.ctaker;

UPDATE available A
SET status='full'
WHERE A.ctaker = NEW.ctaker AND maxpet<=(SELECT COUNT(*) FROM orders O WHERE O.ctaker = A.ctaker AND (O.status='Payment Received' OR O.status='Pending Payment')  AND A.date>=O.sdate AND A.date<=O.edate);

RETURN NEW;
END; $$
LANGUAGE plpgsql;


CREATE TRIGGER update_available_after_payment
AFTER INSERT OR UPDATE ON orders
FOR EACH ROW WHEN (NEW.status = 'Payment Received' OR NEW.status = 'Pending Payment')
EXECUTE FUNCTION update_available();
```

Section VII. Interesting queries

1. Transaction, promote to full-time caretakers:

```
async function promotePartime(username) {
  try {
    await db.query('BEGIN');
    const queryText = 'UPDATE caretakers SET fulltime=true, maxpets=5 WHERE username = $1';
    await db.query(queryText, [username]);
    const queryText2 = '
INSERT INTO
    looksafter(ctaker, price, ptype)
SELECT
    $1,
    0,
    ptype
FROM
    pettypes';
    await db.query(queryText2, [username]);
    const queryText3 = 'SELECT update_price_f(ptype) FROM pettypes';
    await db.query(queryText3);
    const insertAvail = '
INSERT INTO
    available(ctaker, date)
SELECT
    $1,
    dd::date
FROM
    generate_series($2::timestamp, $3::timestamp, \'1 day\'::INTERVAL) dd ON
    CONFLICT DO NOTHING';
    await db.query(insertAvail, [username, '2020-10-01', '2021-12-31']);
    await db.query('COMMIT');
  } catch (err) {
    await db.query('ROLLBACK');
    throw err;
  }
}
```

Explanation of the use of transaction above:

For the promote to full-time caretakers transaction, it consists of two insert operations, one update operation and one call to stored function: update_price_f(ptype). When the PCS admin promotes a part-time caretaker, we first need to update the full-time status and max pet limit in caretakers table. We then need to insert the caretaker data into the looksafter table which stores prices for each pet category and caretakers. Thirdly, we need to call stored function update_price_f(ptype) to update the corresponding prices of each pet category of the caretaker. Lastly, as fulltime caretakers are by default available, we insert the available dates into the available table. The BEGIN command starts a new transaction, and COMMIT updates the database if all operations are successful. If any of the operations encounters error, the database will ROLLBACK and restore to the state before BEGIN. The use of transaction preserves the consistency of the database.

2. **Subquery, get service:**

```
SELECT
    username,
    realname,
    addres,
    fulltime,
    (CASE
        WHEN numrating = 0 THEN -1
        ELSE (sumrating + 0.0)/ numrating
    END) AS rating,
    price * ($3::DATE - $2::DATE + 1) AS totalprice
FROM
    (caretakers C
JOIN looksafter ON
    ctaker = C.username
    AND ptype = $1) NATURAL
JOIN accounts
WHERE
    ($3::DATE-$2::DATE + 1) = (
    SELECT
        COUNT(*)
    FROM
        available A
    WHERE
        A.ctaker = C.username
        AND date >= $2
        AND date <= $3
        AND status = 'available'

        [petcategory, startdate, enddate]
```

Explanation of the use of subqueries in get service query:

In terms of the get service query, the input that user provide is pet category, the intended start date, and the intended end date. The query is designed to return all the care takers who are available to take care of pets of the specific type, given the start date and end date of the service. We have used a subquery in the WHERE clause of the query as shown above. The subquery uses aggregation function COUNT(*) to count the number of rows in the table available in order to count the available days between the start date and end date of a specific caretaker.

3. **Stored Function/Procedure, check clash when admin gets leave information:**

```
CREATE OR REPLACE FUNCTION check_clash(ctakerV VARCHAR, startdateV DATE, enddateV DATE)
RETURNS varchar AS
$$
DECLARE clash varchar = 'true';
BEGIN
if NOT EXISTS (SELECT 1 FROM orders O WHERE O.sdate <= enddateV AND O.edate >= startdateV AND O.ctaker = ctakerV
AND (O.status = 'Payment Received' OR O.status = 'Pending Payment' OR O.status = 'Pending Caretaker Acceptance'))
THEN
clash = 'false';
END IF;
RETURN clash;
END;
$$
language plpgsql;
```

Explanation of the use of stored function:

As shown in the screenshot above, this is one of our uses of stored function. The function check_clash is designed to check whether there is a potential conflict when a caretaker applies for leave and when PCS admin users review ask-for-leave applications. Given the start date and end date that the caretaker want to apply for leave, if there are any orders that the payment has been received, or the payment is still pending, or is still waiting for the caretaker's acceptance, then a time schedule conflict is found and 'true' would be returned, else 'false' would be returned.

Initially, three parameters are passed into the function: ctaker which is a varchar, startdate which is a Date, and enddate which is a Date as well. In the function, we first declare a variable clash which is a varchar. It is initialized to be 'true'. We then use the if statement to find if during this timeslot, does the caretaker has any orders that the payment has been received, or the payment is still pending, or is still waiting for the caretaker's acceptance. If no such order is found, clash is then changed to 'false'. Finally, clash is returned by the function.

```
CREATE OR REPLACE PROCEDURE update_leave(ctakerV VARCHAR, startdateV DATE, enddateV DATE) AS
$$
BEGIN
if check_clash(ctakerV, startdateV, enddateV) = 'false'
THEN
INSERT INTO leave(ctaker, startdate, enddate, clash)
VALUES (ctakerV, startdateV, enddateV, check_clash(ctakerV, startdateV, enddateV));
RAISE NOTICE 'sucessfully done!';
END IF;
END;
$$
language plpgsql;
```

Explanation of the use of stored procedure:

As shown above, this is one of our uses of stored procedure. The procedure update_leave is designed to update the leave table when the caretaker applies for a leave. Given the username of

the caretaker, start date and end date that the caretaker want to apply for leave, if no time conflicts are found, the leave application will be inserted into the leave table, else the application will not be inserted.

Initially, three parameters are passed into the function: ctaker which is a varchar, startdate which is a Date, and enddate which is a Date as well. In the procedure, we check if there are any time schedule conflicts using the check_clash function that we have just mentioned, and if no conflicts are found, we will then insert the leave application into the leave table.

Section VIII. Specifications of technical stack

Frontend

1. Our frontend consists of two Single Page Applications (user site and admin site) based on **Vue.js** (with Vue Router for routing and Vuex for global state management).
2. We use **ElementUI** as the UI framework, **axios** for HTTP requests, **Apache Echarts** for data visualization, **cookies** for persistent local storage.
3. We use **ESLint** as the linter.
4. Both websites are hosted on **Netlify**. The source code is stored on **GitHub**, with **GitHub actions** with custom C++ code to test builds and automate code quality checking.

Backend:

1. Our backend is an API server. It is based on **Node.js** and **Express.js**.
2. Authentication is based on **Json Web Token (JWT)**. OTPs are sent using **Nodemailer**. We set CORS headers to allow all origins, so the frontend can be served from anywhere.
3. **ESLint** is used for linting.
4. The source code is stored on **GitHub** with **GitHub** actions scripts for style checking. Style checking is also done in Git pre-commit hook using **Husky**. The backend is deployed on **Heroku**.

API Documentation:

Our API documentation is created with **Swagger**, hosted on **Netlify**.

Database:

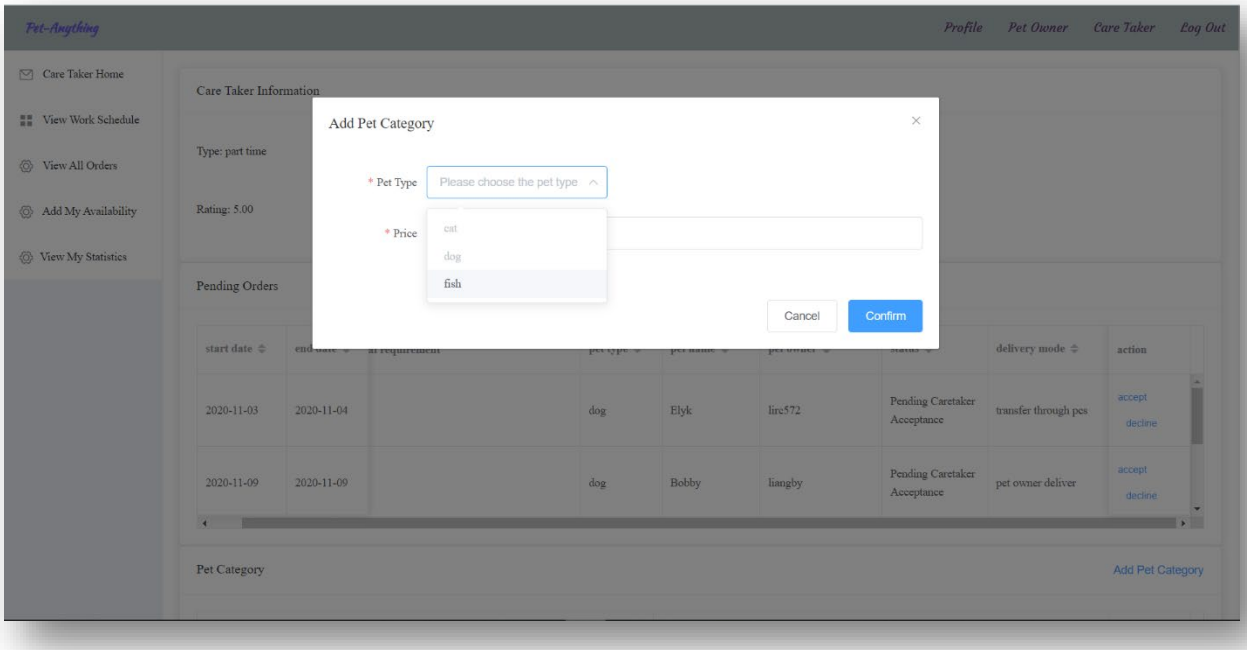
We use **PostgreSQL** as our database. It is hosted on **Heroku**.

Section IX. Screenshots of application

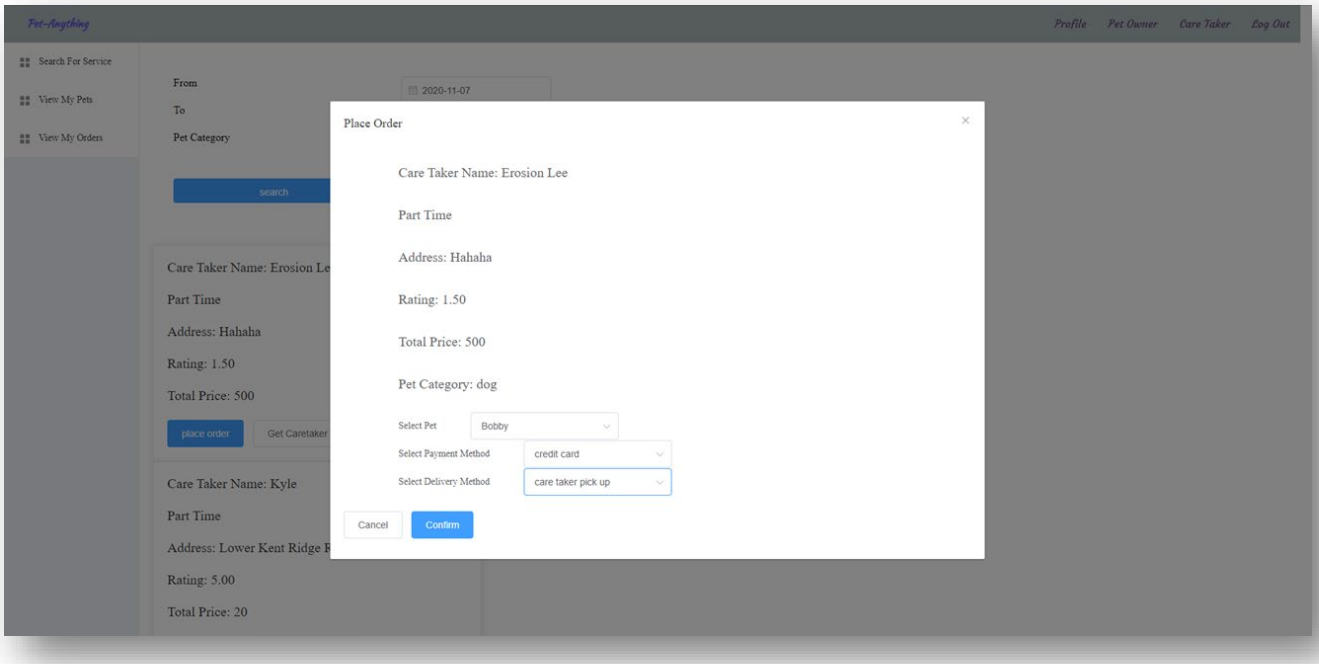
The user interface includes welcome page, signup page, login up, user profile page, pet owner related pages, caretaker related pages and admin related pages. Here are 3 representative screenshots of the application.

The following screenshot shows the dashboard page of admin platform. It visualizes the monthly revenue and the monthly number services provided in the past year, as well as information relevant to the pet owners and caretakers.

The following is a screenshot at the part-time caretaker landing page when he is adding a new pet type with its respective price.



The following is a screenshot when the petowner is specifying payment mode and delivery mode.



Section X. Difficulties

At the beginning of the project, it was difficult to decide on the UI framework and back-end framework. We have learned that Internet search and group brainstorming can be a good starting point, and it is good to select one member to coordinate the entire workflow and pipeline.

During the development, there are often some unexpected error or behavior. In most cases, simple Google searches can find the solution. However, there are still some tricky ones which require some time to solve. Therefore, starting integration tests early is a good option.

It is also stressful to learn node.js along the way of development. Working with an unfamiliar language, many of the implementations and debugging process involves long time searching on Google and stackoverflow. For some quires such as checking 2 * 150 consecutive available days for fulltime caretakers, the logic behind needs careful thinking. We have also learned to adopt different query formats under different situations.