CS2102 Project Report

Topic E: Pet Care

# **Project Group 60**

# SAIF UDDIN MAHMUD [A0170896N] TEO JUN JIE [A0164197W] ALVAREZ NICOLE KEVIN TIMBOL [A0166667M] LIM SING JIE [A0173141R]

# 1 | Division of Responsibility

* Saif Uddin Mahmud was responsible for (1) the relational Schema and ER diagram, (2) all the SQL – including the normal queries, 3 Complex Queries, Complex Triggers and Transactions, and (3) wrote the report, (4) made the dummy data and (5) tested after each integration.
* Teo Jun Jie was responsible for (1) the node-js backend, (2) DevOps, (3) frontend and (4) integration. He also (5) made sure the transactions were done properly and errors on the database handled properly by the server.
* Kevin implemented beautification and helped do final checks before submission.
* Sing Jie implemented beautification and helped do final checks before submission.

It is worth noting that most important decisions - be it the DDL, the program flow or the User Experience – were made by asking for input from everyone.

# 2 | Description of Functionality

The application allows pet owners to search for caretakers for their pets for certain periods of time. Caretakers can advertise their availability publicly with additional information (time range, minimum ask etc), (known as **Services**) while pet owners can browse and **bid** for the services. Caretakers can choose which bid to accept, at which point he is assigned a **task**. At any point in time, the bid can be retracted by the owner. The caretaker can take down the service if some owner doesn’t already have a successful bid on it. The owner can leave a **review** for a task once it is done. Both types of user can send feedback/complaints/**requests** to the system, which are assigned to a **manager** who handles them. There are a few interesting graphs that users can see when they login.   
  
Security was kept in mind when developing the system, and we made sure passwords were properly hashed and unnecessary information was not sent to the client. Triggers were set up to ensure validity of data. However, there are a few security (and by extension , data integrity) loopholes that were not fixed at the time of writing due to lack of time.   
  
It is worth noting that most of the system works on updates and insertions only. Deletions are mostly *soft deletions* (i.e. handled with a specified status number).

# 3 | ER Model

# 4 | Relational Schema

|  |
| --- |
| /\*  \* A USER can be an OWNER or a CARETAKER  \* (i.e Covering Constraint and Overlapping Constraint satisfied)  \*/  **create** **table** USERS (  user\_id bigserial **primary** **key**,  name text **not** **null**,  email text **unique** **not** **null**,  phone **varchar**(20) **unique** **not** **null**,  address json **not** **null**,  password **char**(60) **not** **null**,  created **timestamp** **not** **null** **default** NOW()  );  -- <Sends> collapsed into it  -- status: 0=unattended, 1=attending, 2=solved  **create** **table** REQUESTS (  request\_id bigserial **primary** **key**,  message text **not** **null**,  status **integer** **not** **null** **default** 0 **check** (status>-1 **and** status<3),  created **timestamp** **not** **null** **default** NOW(),  user\_id bigserial **not** **null**,  **foreign** **key** (user\_id) **references** USERS  );  **create** **table** MANAGERS (  manager\_id bigserial **primary** **key**,  email text **unique** **not** **null**,  username text **not** **null**,  password **char**(60) **not** **null**,  phone **varchar**(20) **unique** **not** **null**  );  **create** **table** Handles (  manager\_id bigserial,  request\_id bigserial,  assigned **timestamp** **default** NOW(),  justification text, --by manager  **primary** **key** (manager\_id, request\_id),  **foreign** **key** (manager\_id) **references** MANAGERS,  **foreign** **key** (request\_id) **references** REQUESTS  );  **create** **table** OWNERS (  user\_id bigserial **primary** **key**,  **foreign** **key** (user\_id) **references** USERS  );  **create** **table** ANIMALS (  **type** text **primary** **key**  );  -- <isOfType> collapsed into this  **create** **table** PETS (  pet\_id bigserial **primary** **key**,  name text **not** **null**,  **type** text **not** **null**,  biography text,  born **date** **not** **null** **default** NOW(),  **foreign** **key** (**type**) **references** ANIMALS  );  -- Design consideration. Till and Owns here (instead of Pets as weak entity) because  -- we want the application to be able to handle pet transfers later on without losing information on the  -- pet. Logistics of it has to be solved later, and is beyond the scope of the module.  **create** **table** Owns (  pet\_id bigserial,  owner\_id bigserial,  since **date** **not** **null**,  till **date**,  **primary** **key** (pet\_id, owner\_id),  **foreign** **key** (pet\_id) **references** PETS,  **foreign** **key** (owner\_id) **references** OWNERS(user\_id)  );  **create** **table** CARETAKERS (  user\_id bigserial **primary** **key**,  rating float4 **not** **null** **default** 0,  **foreign** **key** (user\_id) **references** USERS  );  **create** **table** Likes (  caretaker\_id bigserial,  **type** text,  **primary** **key** (caretaker\_id, **type**),  **foreign** **key** (caretaker\_id) **references** caretakers (user\_id),  **foreign** **key** (**type**) **references** ANIMALS  );  -- <Offers> collapsed into this  -- status: 0=retracted, 1=available, 2=taken  **create** **table** SERVICES (  service\_id bigserial **primary** **key**,  caretaker\_id bigserial **not** **null**,  starting **timestamp** **not** **null**,  ending **timestamp** **not** **null** **check** (ending > starting),  status **integer** **not** **null** **default** 1 **check** (status>-1 **and** status<3),  minWage **integer** **not** **null** **check** (minWage > 0),  **foreign** **key** (caretaker\_id) **references** CARETAKERS  );  -- <Places> Collapsed into this  -- status: 0=rejected, 1=pending, 2=success  **create** **table** BIDS (  bid\_id bigserial **primary** **key**,  money **integer** **check** (money>0),  status **integer** **not** **null** **default** 1 **check** (status>-1 **and**  status<3),  starting **timestamp** **not** **null**,  ending **timestamp** **not** **null** **check** (ending > starting),  owner\_id bigserial **not** **null**,  pet\_id bigserial **not** **null**,  service\_id bigserial **not** **null**,  **foreign** **key** (pet\_id) **references** Pets,  **foreign** **key** (owner\_id) **references** Owners,  **foreign** **key** (service\_id) **references** SERVICES  );  -- <Creates> collapsed into this  -- status: 1=upcoming, 2=finished  **create** **table** TASKS (  task\_id bigserial **primary** **key**,  bid\_id bigserial **not** **null** **unique**,  status **integer** **not** **null** **default** 1 **check** (status=1 **or** status=2),  **foreign** **key** (bid\_id) **references** BIDS  );  -- <Gives>, <Receives>, <Has> collapsed into this  **create** **table** REVIEWS (  reviewNum **integer**,  note text,  stars **integer** **not** **null** **check** (stars>=0 **and** stars<=5),  task\_id bigserial **unique** **not** **null**,  caretaker\_id bigserial,  owner\_id bigserial **not** **null**,  **foreign** **key** (task\_id) **references** TASKS,  **foreign** **key** (caretaker\_id) **references** CARETAKERS,  **foreign** **key** (owner\_id) **references** OWNERS,  **primary** **key** (caretaker\_id, reviewNum)  ); |

# 5 | Triggers

3 triggers are listed below with a short description and the code. Please note there are a few other triggers in the code. The list of triggers needed is bigger than the list of triggers implemented, and they were left out of the project in the interest of time.

**A.** The following trigger ensures that **Caretakers are not able to take down a Service they offered if there has already been a successful bid for it (i.e “Task” has been created)**. This trigger also ensures **that Caretakers are not able to accept more than 1 Bid per Service offered**, because they can’t be at two places at the same time.

|  |
| --- |
| **create** **or** replace **function** updateService() **returns** **trigger** **as** $$  **declare** isTask **integer**;  **begin**  **select** **count**(\*) **into** isTask **from** Bids B  **where** B.service\_id=**new**.service\_id **and** status=2;  **if** isTask > 0 **and** **new**.status=0  **then** raise **exception** 'Cannot remove as task exists.'; **return** **null**;  elseif isTask > 0 **and** **new**.status=2  **then** raise **exception** 'Task already exists for this service'; **return** **null**;  **else** **return** **new**; **end** **if**;  **end**; $$ **language** plpgsql;  **create** **trigger** updatingService  **before** **update** **on** services  **for** **each** **row**  **execute** **procedure** updateService(); |

**B.** This next trigger ensures that the **same Caretaker does not offer two concurrent/overlapping Services.**

|  |
| --- |
| **create** **or** replace **function** offerService() **returns** **trigger** **as** $$  **declare** oldStart **timestamp**; oldEnd **timestamp**;  **begin**  **for** oldStart, oldEnd **in** **select** starting, ending **from** services  **where** caretaker\_id=**new**.caretaker\_id **and** status<>0  loop  **if** **new**.starting >= oldStart **and** **new**.starting <= oldEnd  **then** raise **exception** 'Taken/Available service exists with time overlap.';  **return** **null**;  elseif **new**.ending >= oldStart **and** **new**.ending <= oldEnd  **then** raise **exception** 'Taken/Available service exists with time overlap.';  **return** **null**;  elseif **new**.starting <= oldStart **and** **new**.ending >= oldEnd  **then** raise **exception** 'Taken/Available service exists with time overlap.';  **return** **null**;  **else** **return** **new**; **end** **if**;  **end** loop;  **return** **new**;  **end**; $$ **language** plpgsql;  **create** **trigger** offeringService  **before** **insert** **on** services  **for** **each** **row**  **execute** **procedure** offerService(); |
|  |

**C.** The next trigger makes sure that a Bid can be placed by an Owner if the following conditions are met: (1) **the caretaker is compatible with the pet,** (2) **the timeframe bid for lies within the time of service offered,** (3) **the money offered is equal to or greater than the minimum per hour listed by the Caretaker,** (4) **bidding for that service has not already closed .**

|  |
| --- |
| **create** **or** replace **function** placeBid()  **returns** **trigger** **as** $$  **declare** earliest **timestamp**;  latest **timestamp**;  minPerHour **integer**;  preferences text[];  petType text;  compatibility **boolean**;  **begin**  **select** starting, ending, minwage **into** earliest, latest, minPerHour  **from** services **where** service\_id=**new**.service\_id;  **select** **type** **into** petType **from** pets P **where** P.pet\_id=**new**.pet\_id;  **if** **new**.starting < earliest  **then** raise **exception** 'Starts later.'; **return** **null**;  elseif **new**.ending > latest  **then** raise **exception** 'Ends earlier.'; **return** **null**;  elseif (petType **not** **in** (**select** **type** **from** likes  **where** caretaker\_id=(**select** caretaker\_id  **from** Services  **where** service\_id=  **new**.service\_id)))  **then** raise **exception** 'Not compatible with this pet!';  **return** **null**;  elseif minPerHour \* ((**EXTRACT**(EPOCH **FROM** **new**.ending) –  **EXTRACT**(EPOCH **FROM** **new**.starting))/3600.0)>**new**.money  **then** raise **exception** 'Need higher offer.'; **return** **null**;  elseif (**select** status **from** services **where** service\_id=**new**.service\_id)=2  **then** raise **exception** 'Bidding closed.'; **return** **null**;  **else** **return** **new**; **end** **if**;  **end**; $$ **language** plpgsql;  **create** **trigger** placingBid  **before** **insert** **on** Bids  **for** **each** **row**  **execute** **procedure** placeBid(); |

# 6 | Interesting Queries

There are 3 interesting non-trivial queries our website supports. All of them show data on graphs in the frontend so different stakeholders of the website (caretakers, owners and managers) may use the data to their benefit.

1. For every Task that the Caretaker does, he/she earns a certain amount of money per hour. The following query gives us, for every Caretaker in the system, the average earned per hour averaged over the entire month for 12 different months. Only the manager is privy to such information.

**select** S.caretaker\_id,

**extract**(**month** **from** B.starting)::**integer** **as** **month**,

**coalesce**(**avg**(money/((**EXTRACT**(EPOCH **FROM** B.ending)

- **EXTRACT**(EPOCH **FROM** B.starting))/3600.0)), 0) **as** money

**from** Bids B **join** Tasks T **on** (T.bid\_id=B.bid\_id) **join** Services S **on** (B.service\_id=S.service\_id)

**group** **by** S.caretaker\_id, **extract**(**month** **from** B.starting)

**order** **by** **extract**(**month** **from** B.starting);',

1. The following query helps the caretaker gauge his earnings over time in while working with the platform. The query uses a window function with the aggregate to show the cumulative income by day on the caretaker dashboard.

**select** **extract**(**year** **from** B.starting)::**integer** **as** **year**,

**extract**(**month** **from** b.starting)::**integer** **as** **month**,

**extract**(**day** **from** b.starting)::**integer** **as** **day**, **sum**(**sum**(money))

**over** (**order** **by** **extract**(**year** **from** B.starting),

**extract**(**month** **from** b.starting),

**extract**(**day** **from** b.starting))

**from** Bids B **join** Tasks T **on** (T.bid\_id=B.bid\_id) **join** Services S **on** (B.service\_id=S.service\_id)

**where** S.caretaker\_id=$1 **group** **by** **extract**(**year** **from** B.starting),

**extract**(**month** **from** b.starting),

**extract**(**day** **from** b.starting)

**order** **by** **sum** **desc** offset $2 **limit** $3;

1. This last interesting query shows the demand by hour during the day, for the day specified. This lets caretakers and owners make better judgement when offering services and bidding, and use the supply-demand dynamics.

**select** **extract**(DOW **from** starting) **as** **day**,

**extract**(**hour** **from** starting)::**integer** **as** **hour**, **count**(\*)::float4/(**select** **count**(\*) **from** Bids

**where** **extract**(DOW **from** starting)=$1) **as** ratio

**from** bids   
**where** **extract**(DOW **from** starting)=$1   
**group** **by** **day**, **hour**;

# 7 | Framework and Software Tools Used

The application was made with an **MVC framework** and **Node.js runtime**. The notable packages used were: bcrypt, node-postgres, passport. We also made use of Bootstrap for a better frontend, along with jquery and ajax. The RDBMS was **PostgreSQL**. Development and Testing was done on different platforms, but a Github repository was used for Version Control and Project Management.

# 8 | How to Run

1. You can download code60.zip and unzip it at a location of interest.
2. You can then call “npm install” to install all the packages and their dependencies.
3. You’ll have to have a postgres server running in the background with the name “cs2102\_petcare\_trial”.
4. Take the code in database/init\_petcare.sql and run it on the database you created.   
   This has the Relational Schema, Triggers and dummy data.
5. You’ll need a .env file having the following lines of code. (please note that you may need to slightly change and/or give it the username and password as well)

*DATABASE\_URL=postgres://localhost:5432/cs2102\_petcare\_trial*

*SESSION\_SECRET=secret*

You can now run the code with “npm start”

If at any stage you have problems running the code, please do not hesitate to send your queries to [saifum@u.nus.edu](mailto:saifum@u.nus.edu) or create an issue at <https://github.com/cs2102team60petcare/server>.

We will get back to you as soon as possible.

# 9 | Representative Screenshots

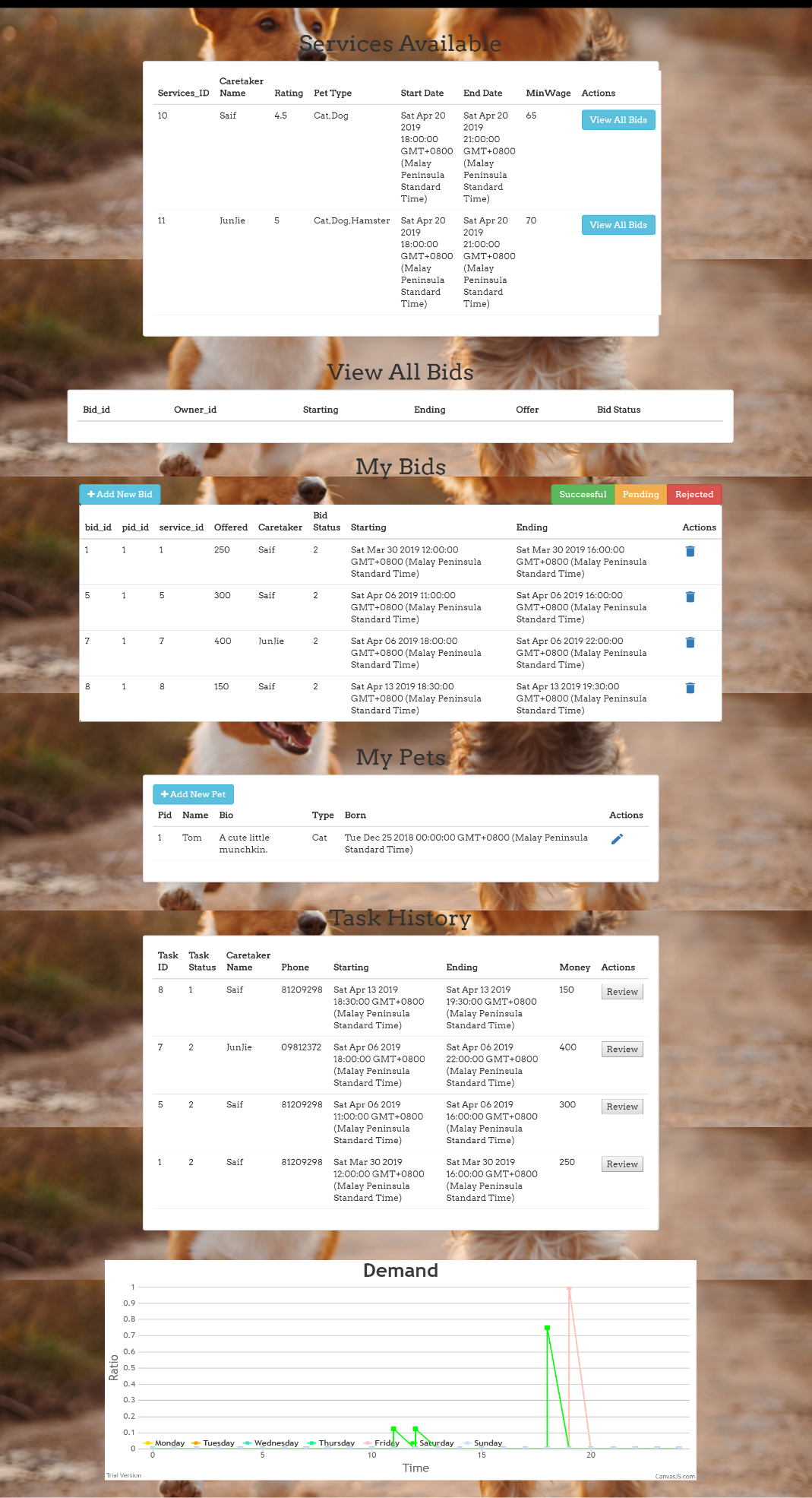


Figure 1: Owner profile page

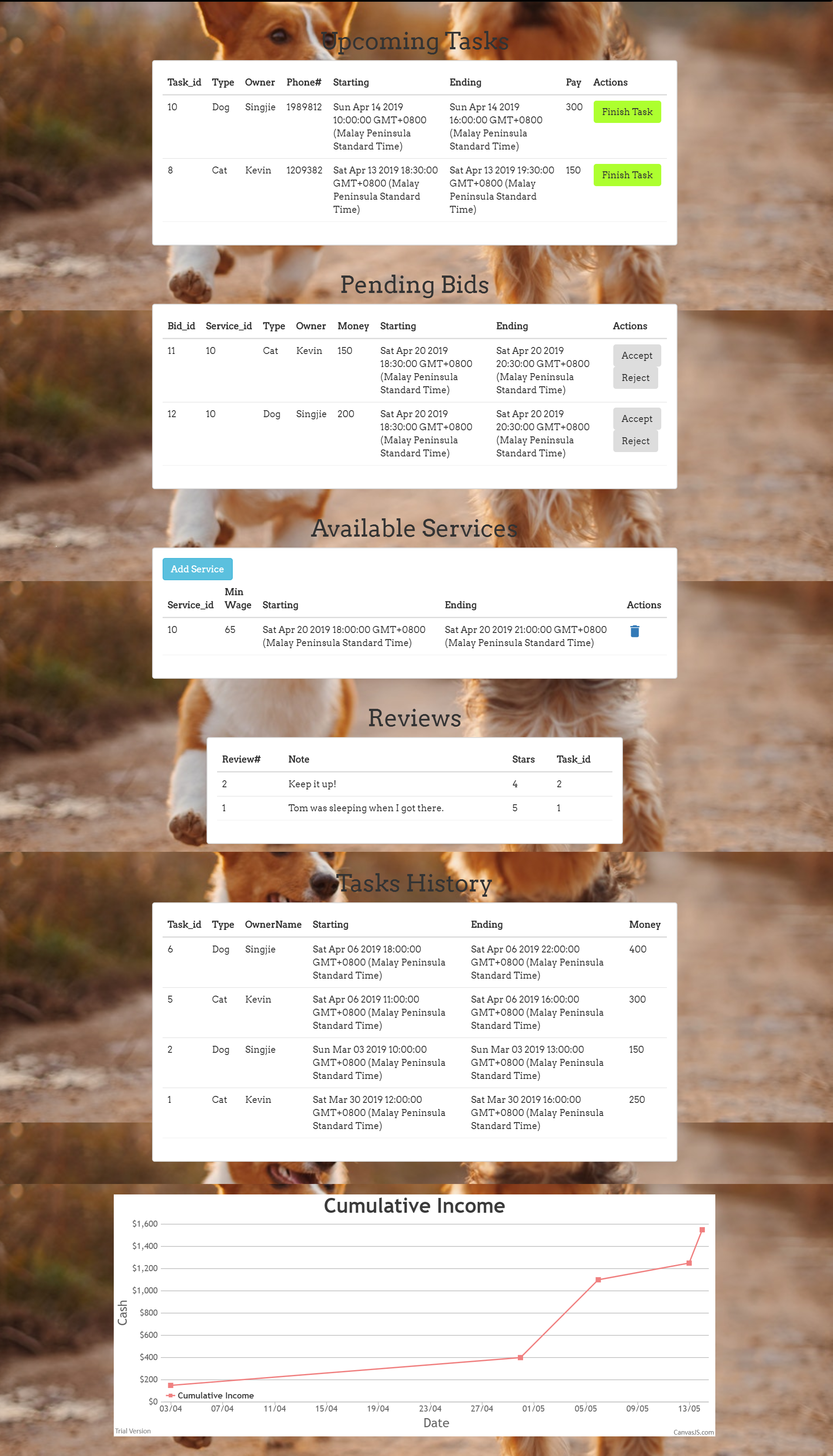


Figure 2: Caretaker Profile Page

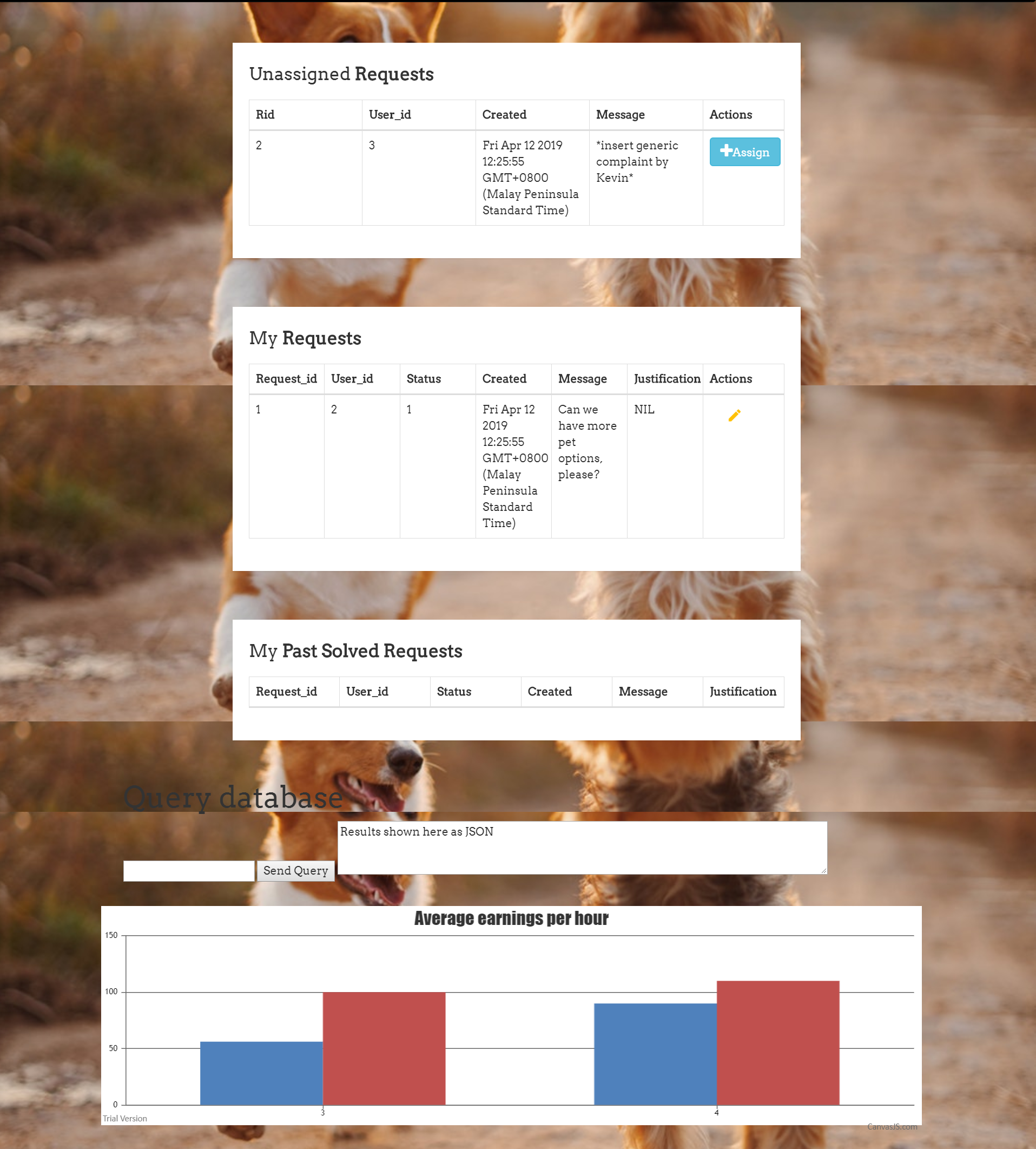


Figure 3: Manager profile page

# 10 | Difficulties Encountered & Lessons Learnt

The team faced numerous difficulties when doing the project. The major difficulty came when we designed the program flow. We lost track of the project specifications and made a Schema that was not compatible with the requirements. As a result, we had to scrape a week’s worth of work when we realize and start from scratch. The lesson learnt is to always have the project specification in hand when making major decisions to check if we’re missing something.

The other significant difficulty we faced was of a high workload. We took the project a bit too seriously, and made decisions depending on how an actual business/startup would do it. By the time we realized this, it was too late and we had done a lot of work in that direction. This proved to strain our time. The lesson learnt is to start lean and get an MVP out, and have a working product to show every step of the way.

Another major difficulty was because 2 out of 4 people in the group did not have any web development experience, the learning process was quite slow. As such, testing the software fully was left off till the last moment as we frantically put 3 all-nighters to get the project done.

--------------------------------------------------------------------------------------------------------------------------------------

# Appendix A: Queries and Transactions

The Queries and Transactions are given below for reference. You can also open **database/queries.js** in your favourite IDE. Please note that transactions are implemented with client.query and Promise (asynchronous await) as documented in node-pg. You can spot a transaction below by looking for a number after the variable name (for example, retractBidUpdate1-4 are supposed to be done in a single transaction)

module.exports = {

/\* ACCESS RELATED QUERIES \*/

loginQuery: 'SELECT user\_id, name , email, password FROM users WHERE email=$1;', // used for login

loginManagerQuery: 'SELECT manager\_id, email, password FROM MANAGERS where email=$1;',

deserializeUserQuery: 'SELECT user\_id, name, email FROM users WHERE user\_id=$1;', // used for sessions

deserializeManagerQuery: 'select \* from managers where email=$1;',

isOwnerOrCaretakerQuery: 'SELECT 1 from OWNERS natural join USERS where email=$1 UNION SELECT 2 from CARETAKERS natural join USERS where email=$1;',

userExistsQuery: 'SELECT user\_id FROM users WHERE email=$1;', // todo: @ jj used as check before signup

// WARNING: Use signUpUserInsert in a Transaction with one of the next two

signupUserInsert: 'INSERT INTO users (name, email, phone, address, password) VALUES ($1, $2, $3, $4, $5) RETURNING \*;',

signupOwnerInsert: 'INSERT INTO owners (user\_id) VALUES ($1);',

signupCareTakerInsert: 'INSERT INTO caretakers (user\_id) VALUES ($1);',

careTakerLikesInsert: 'INSERT INTO likes (caretaker\_id, type) VALUES ($1, $2);',

careTakerLikesRemove: 'DELETE FROM likes where caretaker\_id=$1;',

// WARNING: Use signupPetInsert with the next query in a transaction

signupPetInsert: 'INSERT INTO pets (name, type, biography, born) VALUES ($1, $2, $3, $4) RETURNING \*;',

ownsPetInsert: 'INSERT INTO owns (pet\_id, owner\_id, since) VALUES ($1, $2, $3);',

/\* PROFILE RELATED QUERIES \*/

fullUserProfileQuery: 'SELECT user\_id, name, email, phone, address, created FROM users where user\_id=$1;',

petProfileQuery: 'SELECT \* from PETS where pet\_id=$1;',

userProfileUpdate: 'UPDATE Users SET name=$1, email=$2, phone=$3, address=$4 WHERE user\_id=$5 RETURNING \*;',

// WARNING: Make sure Password is hashed with bcrypt when updating

userPasswordUpdate: 'UPDATE Users SET password=$1 WHERE user\_id=$2;',

petProfileUpdate: 'UPDATE Pets SET name=$1, biography=$2 WHERE pet\_id=$3;',

getMyUpcomingTasksQuery: 'SELECT task\_id, P.type, P.name as petname, U.name as ownername, U.phone, B.starting, B.ending, B.money FROM Tasks T JOIN Bids B on (T.bid\_id=B.bid\_id) JOIN Services S on (B.service\_id=S.service\_id) NATURAL JOIN Caretakers C NATURAL JOIN Pets P JOIN Users U on (B.owner\_id=U.user\_id) WHERE C.user\_id=$1 and T.status=1 ORDER BY S.starting desc;',

getMyTaskHistoryQuery: 'SELECT task\_id, P.type, P.name as petname, U.name as ownername, B.starting, B.ending, B.money FROM Tasks T join Bids B on (T.bid\_id=B.bid\_id) join Services S on (B.service\_id=S.service\_id) JOIN Caretakers C on (S.caretaker\_id=C.user\_id) NATURAL JOIN Pets P JOIN Users U on (B.owner\_id=U.user\_id) WHERE C.user\_id=$1 and T.status=2 ORDER BY S.starting desc;',

getMyTaskHistoryAsOwnerQuery: 'SELECT task\_id, T.status, P.name as petname, U.name as caretakername, U.phone, B.starting, B.ending, B.money FROM Tasks T join Bids B on (T.bid\_id=B.bid\_id and B.owner\_id=$1) join Services S on (B.service\_id=S.service\_id) JOIN Caretakers C on (S.caretaker\_id=C.user\_id) NATURAL JOIN Pets P JOIN Users U on (S.caretaker\_id=U.user\_id) ORDER BY S.starting desc;',

finishTaskUpdate: 'UPDATE Tasks SET status=2 where task\_id=$1;',

getMyReviews: 'SELECT \* from REVIEWS where caretaker\_id=$1 order by reviewnum desc;',

getMyPetsQuery: 'SELECT \* FROM Pets P NATURAL JOIN Owns O WHERE O.owner\_id=$1;',

// Use when Caretaker is offering new service

// Note: One task per service, even if the caretaker has free time.

// Triggers offeringService

offerServiceInsert: 'INSERT INTO services (caretaker\_id, starting, ending, minWage) VALUES ($1, $2, $3, $4);',

serviceHistoryQuery: 'SELECT S.service\_id, T.task\_id, S.status, S.starting, S.ending, T.status, S.minWage, B.money, B.owner\_id, B.pet\_id ' +

'FROM Services S LEFT OUTER JOIN (Bids B join Tasks T on B.bid\_id=T.bid\_id) on S.caretaker\_id=$1 and (B.service\_id=S.service\_id) ' +

'ORDER BY (starting) OFFSET $2 LIMIT $3;',

getMyAvailableServicesQuery: 'SELECT \* FROM SERVICES WHERE caretaker\_id=$1 and status=1;',

/\* BIDS RELATED QUERIES \*/

getPendingBidsForMeQuery: 'select B.bid\_id, S.service\_id, P.type, U.name, B.starting, B.ending, B.money ' +

'from caretakers C join services S on (C.user\_id=S.caretaker\_id) join Bids B on (B.service\_id=S.service\_id) ' +

'join Pets P on (B.pet\_id=P.pet\_id) join Users U on (B.owner\_id=U.user\_id) ' +

'where S.status=1 and B.status=1 and S.caretaker\_id=$1;',

seeBidsForServiceQuery: 'SELECT B.owner\_id, B.status, B.bid\_id, S.service\_id, B.money, B.starting, B.ending FROM Bids B JOIN Services S on (B.service\_id=S.service\_id) ' +

'WHERE S.service\_id=$1 ORDER BY B.money desc;',

seeMyBidsQuery: 'SELECT B.bid\_id, B.pet\_id, B.service\_id, B.money, B.status, U.name, B.starting, B.ending FROM Bids B JOIN Services S on (B.service\_id=S.service\_id) join Users U on (U.user\_id=S.caretaker\_id) where B.owner\_id=$1;',

getAllBids: 'SELECT \* FROM Bids',

// Triggers placingBid

placeBidInsert: 'INSERT INTO Bids (starting, ending, money, owner\_id, pet\_id, service\_id) VALUES ($1, $2, $3, $4, $5, $6);',

// Use when Caretaker is removing a service (can't do it if already a task)

// NOTE: triggers updateService

removeServiceUpdate1: 'UPDATE Services SET status=0 WHERE service\_id=$1;',

removeServiceUpdate2: 'UPDATE Bids SET status=0 WHERE service\_id=$1;',

/\* REQUEST or SUPPORTTICKET RELATED QUERIES \*/

sendRequestInsert: 'INSERT INTO Requests (message, user\_id) VALUES ($1, $2);',

getUnassignedRequests: 'SELECT \* FROM Requests WHERE status=0;',

getSolvedRequestsAssignedToMe: 'SELECT \* FROM Requests NATURAL JOIN Handles WHERE manager\_id=$1 and status=2 ORDER by status OFFSET $2 LIMIT $3;',

getUnresolvedRequestsAssignedToMe: 'SELECT \* FROM Requests NATURAL JOIN Handles WHERE manager\_id=$1 and status=1 ORDER by status OFFSET $2 LIMIT $3;',

getRequestsAssignedToMeWithFilters: 'SELECT \* FROM Requests NATURAL JOIN Handles WHERE manager\_id=$1 and request\_id=$2;',

rejectBidUpdate: 'UPDATE Bids SET status=0 WHERE bid\_id=$1;',

// Trigger sendReview()

sendReviewInsert1: 'INSERT INTO Reviews (stars, note, task\_id, caretaker\_id, owner\_id) VALUES ($1, $2, $3, $4, $5);',

sendReviewInsert2: 'UPDATE Tasks set status=2 where task\_id=$1;', // $1 = $3 from above

sendReviewInsert3: 'UPDATE Caretakers SET rating=((select sum(stars)::float4 from reviews where caretaker\_id=$1)/(select count(\*) from reviews where caretaker\_id=$1)) where user\_id=$1;', // $4 = $1 from above

taskCompletedupdate: 'UPDATE Tasks SET status=2 where task\_id=$1;',

// Use when Owners want to retract a bid (CAN do even if already a task)

// Again, you can only remove and add bids, no edits.

// Trigger deletingTask ensures finished tasks/bids can't be deleted (both soft and hard)

retractBidUpdate1: 'UPDATE Bids SET status=0 WHERE owner\_id=$1 and bid\_id=$2;',

retractBidUpdate2: 'UPDATE Bids SET status=1 WHERE owner\_id<>$1 and service\_id=$2;',

retractBidUpdate3: 'UPDATE Services SET status=1 WHERE service\_id=$1;',

retractBidUpdate4: 'DELETE FROM Tasks where bid\_id=$1;',

// Use when caretaker accepts a bid

// make sure no other accepted bids present

// Triggers updateService

acceptBidUpdate1: 'UPDATE Services SET status=2 WHERE service\_id=$1;',

acceptBidUpdate2: 'UPDATE Bids SET status=2 WHERE bid\_id=$1',

acceptBidUpdate3: 'UPDATE Bids SET status=0 WHERE bid\_id<>$1 and service\_id=$2;',

acceptBidUpdate4: 'INSERT INTO Tasks (bid\_id) VALUES ($1);',

// Complex Query 1

// Gives you the average of (average made per hour) grouped by month, for each caretaker

// on the manager dashboard

perHourAverageByMonthQuery: 'select S.caretaker\_id, extract(month from B.starting)::integer as month, coalesce(avg(money/((EXTRACT(EPOCH FROM B.ending) - EXTRACT(EPOCH FROM B.starting))/3600.0)), 0) as money from Bids B join Tasks T on (T.bid\_id=B.bid\_id) join Services S on (B.service\_id=S.service\_id) group by S.caretaker\_id, extract(month from B.starting) order by extract(month from B.starting);',

// Complex Query 2

// Shows the caretaker the cumulative earning (by day)

// on the caretaker\_dashboard

perDayCumulativeSumQuery: 'select extract(year from B.starting)::integer as year, extract(month from b.starting)::integer as month, extract(day from b.starting)::integer as day, sum(sum(money)) over (order by extract(year from B.starting), extract(month from b.starting), extract(day from b.starting)) from Bids B join Tasks T on (T.bid\_id=B.bid\_id) join Services S on (B.service\_id=S.service\_id) where S.caretaker\_id=$1 group by extract(year from B.starting), extract(month from b.starting), extract(day from b.starting) order by sum desc offset $2 limit $3;',

// Complex Query 3

// Shows the demand ratio by hour for days {1..7} (whichever we call)

ratioOfBidsByHourByDay: 'select extract(DOW from starting) as day, extract(hour from starting)::integer as hour, ' +

'count(\*)::float4/(select count(\*) from Bids where extract(DOW from starting)=$1) as ratio ' +

'from bids where extract(DOW from starting)=$1 group by day, hour;',

// Do inside a transaction

assignRequestToMe1: 'UPDATE Requests SET status=1 where request\_id=$1;',

assignRequestToMe2: 'INSERT INTO Handles (manager\_id, request\_id) VALUES ($1, $2);',

// Do inside a transaction

requestSolvedUpdate1: 'UPDATE Requests SET status=2 where request\_id=$1;',

requestSolvedUpdate2: 'UPDATE Handles SET justification=$1 where request\_id=$2;',

// ----------------------- TESTED UNTIL HERE --------------------------------//

// use the base string + other strings

// use for the filter in the owner\_home page

// add num accordingly at the end of every non-base string (or $)

searchAvailableServicesBase: "SELECT S.service\_id, S.starting, S.ending, S.minWage, U.name, C.rating, " +

"array\_to\_string(array\_agg(distinct L.type), ',') as petType " +

"FROM SERVICES S JOIN Caretakers C on (S.caretaker\_id=C.user\_id) JOIN USERS U on (U.user\_id=C.user\_id) join Likes L on (C.user\_id=L.caretaker\_id) WHERE status=1 GROUP BY S.service\_id, U.name, C.rating",

serachAvailableServicesStarting: ' and S.starting>=$',

searchAvailableServicesEnding: ' and S.ending<=$',

searchAvailableServicesCaretaker: ' and U.name=$',

searchAvailableServiesPetType1: ' and $',

searchAvailableServicePetType2: '= ANY(SELECT type FROM Likes L2 where L2.caretaker\_id=S.caretaker\_id)',

getCareTakerIDByName: 'SELECT C.user\_id FROM caretakers C join users U on C.user\_id = U.user\_id where U.name = $1'

}

# Appendix B: Other triggers

We implemented more than 3 triggers and the triggers below are interesting (and complex!) too. We’re referencing them here. Please note that even with the 6 triggers, the data is open to abuse in various manners. We could not enforce very defensive coding because of (1) lack of time, and (2) performance issues.

**D.** The next trigger ensures that **(1) the review numbers increases and (2) a Task is finished before the allowing the Owner to review a task .**

|  |
| --- |
| create or replace function sendReview()  returns trigger as $$  declare lastNum integer; endTime timestamp;  begin      select coalesce(max(reviewnum), 0) into lastNum from reviews where caretaker\_id=new.caretaker\_id;      select ending into endTime from tasks natural join bids where task\_id=new.task\_id;      if endTime > NOW() then raise exception 'Wait till the task is over to send review.'; return null;      else new.reviewnum = lastNum+1; return new;      end if;  end; $$ language plpgsql;  create trigger sendingReview  before insert on Reviews  for each row  execute procedure sendReview(); |

**E.** The next trigger ensures that **a Task can only be deleted if the task is finished.**

|  |
| --- |
| create or replace function deleteTask()  returns trigger as $$ begin      if (select ending from Bids where bid\_id=old.bid\_id) < NOW() then raise exception 'Cant delete if task is finished'; return null;      elseif old.status=2 then raise exception 'Cant delete as task is finished.'; return null;      else return new; end if;  end; $$ language plpgsql;  create trigger deletingTask  before delete on Tasks  for each row  execute procedure deleteTask(); |

**F.** The next trigger ensures that **Caretaker can only the end the Task when the Task is finished.**

|  |
| --- |
| create or replace function finishTask()  returns trigger as $$ begin      if new.status=2 and (select ending from Bids where bid\_id=old.bid\_id) > NOW() then raise exception 'Wait till the task is over!'; return null;      else return new; end if;  end; $$ language plpgsql;  create trigger finishingTask  before Update on Tasks  for each row  execute procedure finishTask(); |