CS2102 Project Report

Topic E: Pet Care

# **Project Group XX**

# Saif Uddin Mahmud [A0170896N] Teo Jun Jie [] Kevin Alvarez [] Kalikala Singjie []

# 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) co-wrote the report.
* Teo Jun Jie was responsible for (1) the node-js backend and (2) DevOps. He helped set up all the routes, required libraries and plugins, data pipeline to the frontend and (3) made sure the transactions were done properly and errors on the database handled properly by the server.
* Kevin Alvarez implemented the frontend and co-wrote the report
* Kalikala Singjie implemented the frontend and generated the dummy data.

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 (3NF Considerations)

/\*

\* A USER has to be an OWNER or a CARETAKER, but not both

\*/

**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**,

death **date**,

**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** **notice** 'Cannot remove as task exists.'; **return** **null**;

**elseif** isTask > 0 **and** **new**.status=2

**then** **raise** **notice** '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** **notice** 'Taken/Available service exists with time overlap.'; **return** **null**;

**elseif** **new**.ending >= oldStart **and** **new**.ending <= oldEnd

**then** **raise** **notice** 'Taken/Available service exists with time overlap.'; **return** **null**;

**elseif** **new**.starting <= oldStart **and** **new**.ending >= oldEnd

**then** **raise** **notice** 'Taken/Available service exists with time overlap.'; **return** **null**;

**else** **return** **new**;

**end** **if**;

**end** **loop**;

**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**;

preferences **text**[];

likesType **text**;

petType **text**;

compatibility **boolean**;

**begin**

**select** starting, ending **into** earliest, latest **from** services **where** service\_id=**new**.service\_id;

**select** likes **into** preferences **from** caretakers **natural** **join** services **where** service\_id=**new**.service\_id **limit** 1;

**select** **type** **into** petType **from** pets **where** pet\_id=**new**.pet\_id;

compatibility:= **false**;

-- ToDO petTypeCompatibility @Psyf

-- ToDO meetsMinWage per hous @ Psyf

**if** **new**.starting < earliest **then** **raise** **notice** 'Starts later.'; **return** **null**;

**elseif** **new**.ending > latest **then** **raise** **notice** 'Ends earlier.'; **return** **null**;

-- elseif compatibility=false then raise notice 'Not in pet preference.'; return null;

**elseif** (**select** status **from** services **where** service\_id=**new**.service\_id)=2 **then** **raise** **notice** 'Bidding closed.'; **return** **null**;

-- or ^ == 0

**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, her 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, **date\_trunc**('month', B.starting),

**coalesce**(**avg**(**money**/((**EXTRACT**(EPOCH **FROM** B.ending) - **EXTRACT**(EPOCH **FROM** B.starting))/3600.0)), 0)

**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, **date\_trunc**('month', B.starting)

**order** **by** **date\_trunc**('month', B.starting) **desc**;

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** **date\_trunc**('day', B.starting), **sum**(**sum**(**money**)) **over** (**order** **by** **date\_trunc**('day', 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** **date\_trunc**('day', B.starting)

**order** **by** **date\_trunc**('day', B.starting) **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**(**hour** **from** starting)::**integer** **as** **hour**,

**count**(\*)::**float4**/(**select** **count**(\*)

**from** Bids **where** status=2 **and** **extract**(DOW **from** starting)=$1)

**from** bids **where** status=2 **and** **extract**(DOW **from** starting)=$1

**group** **by** **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. 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 | Representative Screenshots

# 9 | 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.