April 12th, 2019

CS2102 Project Report

Team 26 - Topic A: Task Sourcing

Philip Garabandic A0192971U
Lim Ming En A0155160N
Iryna Luchak A0196092X
Maxim Salikhov A0195911Y

Table of Contents

1. Project Responsibilities

Philip Garabandic	● Welcome Page ● Administrator Panel
Lim Ming En	● Account Management/Profile Management ● Triggers
Iryna Luchak	● Booking System ● DDL Statements
Maxim Salikhov	● Authentication/registration System ● Worker Panel ● Referrals System ● Review System ● Services Search System ● Triggers/Schema Design

2. Software Tools and Frameworks

Platform: Node.js - JavaScript runtime built on Chrome's v8 JavaScript engine
Database: PostgreSQL v11.1
Web Server Framework:

- Templating Engine: EJS
- Authentication Middleware: Passport.js
- Time/Date Framework: Moment.js

CSS Framework: Bootstrap 4
Programming Languages Used: JavaScript, HTML, CSS, SQL
Software Used: Visual Studio Code, DBeaver, GitHub/Git, PSQL

3. Requirements and Functionality

3.1 General Functionality

Administrator Panel

The administrator panel should enable the website administrators to control which workers are approved to create and offer new services, as well as have the ability to monitor, and disable different services. When a worker initially creates an account, the administrator should be able to quickly see which worker accounts require approval, and be able to do so through the administrator panel. For services, administrators should be able to monitor different services, and disable any services they are monitoring. Each service can be monitored by at most one administrator.

Account Management

Every type of account should be able to update their first name, last name, and password. Additionally, if their account is of only one type, they should be able to add the other account type to their account. However, they will not be able to remove a type from their account, as there are possibly other records tied to that particular account type (e.g. services tied to a worker account). If the account has type user, the user phone number and address on file should be able to be updated. If the account has type worker, the work phone number should be able to be updated. If an account is both a user and a worker, the phone numbers for the two account types should be independent.

Booking System

The user account type should be able to make a booking for a particular service, and should be required to specify the time of service that is within the worker's availability, as well as a valid payment method. The user will also be able to see a summary of their upcoming bookings, and have the ability to cancel future bookings, and leave reviews for past bookings.

Review System

After the end time of a booking, the user should be able to leave a review that includes text and an integer rating from 1 to 5 for the booking. Each booking has to be associated with either no review, or a unique review instance from other bookings. The worker should be able to view a summary of their performance, as well as the text of their reviews (through the worker panel). There can only be one review per booking.

Authentication/Registration System

Website visitors should be able to create an account with the website, with a unique email that is not already registered on the website. During the process of registration, they will be able to specify which account types they can create: worker and user. They can choose one of the two, or both. Once registration is complete, the new account should immediately be able to log into the website using the password specified during registration.

Worker Panel

Through the worker panel, the worker account type should be able to manage their bookings, services offered, availability, and view a summary of the reviews from their previous bookings. There should be an ability to view the particulars of a booking, as well as cancel a booking that is still in the future. Workers should be able to create and update the details of the services they offer, as well as remove any services they no longer wish to offer. A worker should be able to specify their availability for the future, which is used in the services search system. If the time interval the worker specifies overlaps with an existing availability time intervals they had, the two intervals should be merged into one. Workers should also be able to remove and update time intervals of their availability. On the

reviews summary page, workers should be able to see their average rating, as well as an average rating by category and by service area. They should also be able to see the latest text of the reviews they received.

Referrals System

Every account type should be able to submit an email they want to refer to the website. When the email the referral was "sent" to registers for the website while specifying a valid referrer, they will receive a 25% discount code they can use on booking a service. The original referrer should then also receive a 25% discount code they can use. An account cannot refer itself, and an account cannot be referred more than once. Each discount code can be used on at most one booking.

Services Search System

Every website visitor should be able to see a list of services that have future availability, have approved workers offering them, and that are not disabled by website administrators. Website visitors should be able to filter this list by service name, price, category, area where the service is offered, and by the average rating of previous bookings for that service. Additionally, there should be an option to view additional details about each service such as its description. Accounts with type user should have an option to book any of the services presented in their search results.

3.2 Additional Constraints

- If a worker account is approved, it must have an administrator associated with it; if a worker account is not yet approved, it must not have an administrator associated with it
- Discounts may either have an amount of discount or a percentage discount associated to them, but never both

4. Database Schema

```
create table accounts (
  id          serial        primary key,
  email       varchar(254)  unique not null,
  salt        varchar(32)   not null,
  hash        text          not null,
  firstName   varchar(40)   not null,
  lastName    varchar(40)   not null
);
```

...continued on next page...

```
create table users (
  id          serial        primary key references accounts(id) on
delete cascade,
  phone       varchar(40),
  address     text
);

create table refers (
  referrerId  integer       not null references accounts(id),
  email       varchar(254)  unique not null,
  primary key(referrerId, email)
);

create table workers (
  id          serial        primary key references accounts(id) on
delete cascade,
  phone       varchar(40)
);

create table admins (
  id          serial        primary key references accounts(id) on
delete cascade
);

create table categories (
  catId       serial        primary key,
  name        varchar(40)   not null unique
);

create table cityregions (
  regionId    serial        primary key,
  name        varchar(40)   not null unique
);

create table services (
  serviceId   serial        primary key,
  name        varchar(40)   not null,
  description text          not null default 'N/A',
  price       numeric       not null check (price > 0),
  workerId    integer       not null references workers(id),
  catId       integer       not null references categories(catId),
  regionId    integer       not null references cityregions(regionId)
);
```

…continued on next page…

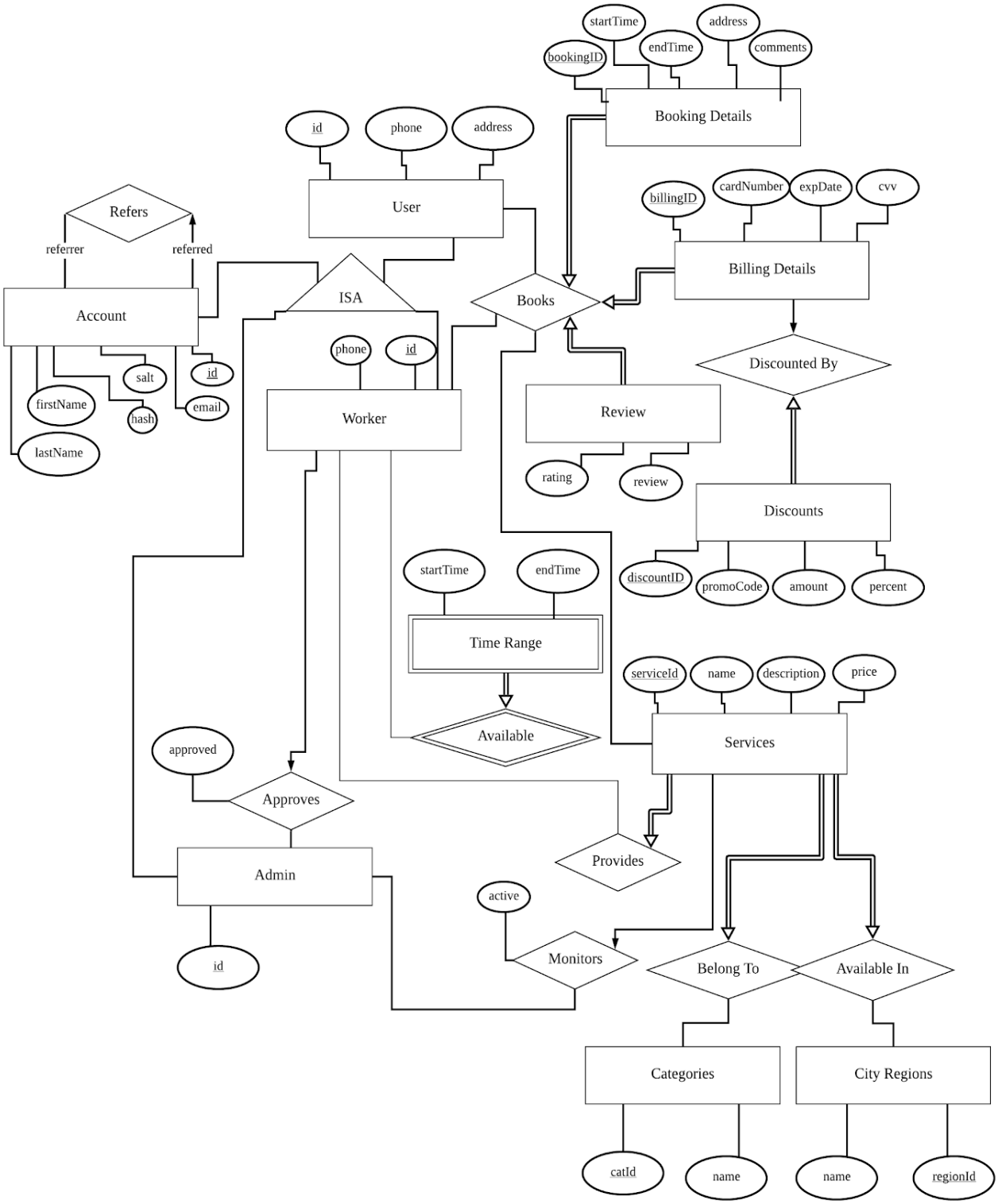
```
create table approves (
  workerId    integer       primary key references workers(id),
  approved    boolean       not null default FALSE,
  adminId     integer       references admins(id),
  check ((approved = true and adminId is not null) or (approved =
false and adminId is null))
);
create table monitors (
  serviceId   integer       primary key references
services(serviceId),
  active      boolean       not null default TRUE,
  adminId     integer       not null references admins(id)
);

create table availability (
  workerId    integer       not null references workers(id),
  startTime   timestamp     not null check (startTime > NOW()),
  endTime     timestamp     not null check (endTime > NOW()),
  primary key(workerId, startTime, endTime),
  check(endTime > startTime)
);

create table discounts (
  discountId  serial        primary key,
  promoCode   varchar(12)   not null unique,
  amount      numeric       check (amount > 0),
  percent     numeric       check (percent > 0 and percent < 100),
  check ((amount is null and percent is not null) or (amount is not
null and percent is null))
);

create table billingdetails (
  billingId   serial        primary key,
  cardNumber  varchar(16)   not null,
  expDate     varchar(5)    not null,
  cvv         varchar(4)    not null,
  discountId  integer       unique references discounts(discountId)
);
```

…continued on next page…

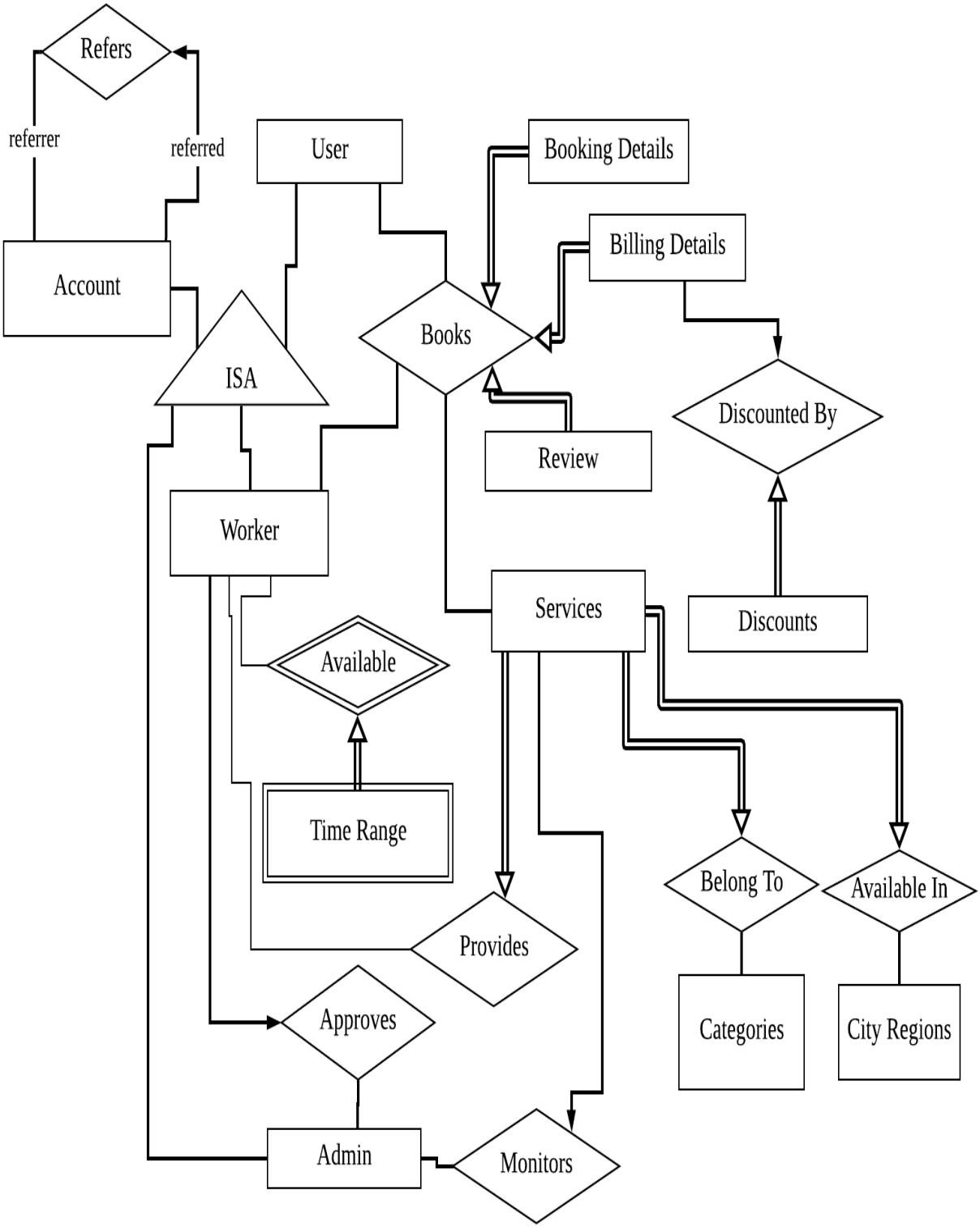
```
create table reviews (
  reviewId    serial        primary key,
  rating      integer       not null check (rating >= 1 and rating <=
5),
  review      text
);

create table bookingdetails (
  bookingId   serial        primary key,
  startTime   timestamp     not null,
  endTime     timestamp     not null,
  address     text          not null,
  comments    text          not null default 'N/A',
  billingId   integer       unique not null references
billingdetails(billingId) on delete cascade,
  userId      integer       not null references users(id),
  workerId    integer       not null references workers(id),
  serviceId   integer       not null references services(serviceId),
  reviewId    integer       unique references reviews(reviewId)
);
```

5. ER Diagram

5.1 ER Diagram with Attributes



5.2 ER Diagram without Attributes



6. Non-trivial Constraints

6.1 Worker Availability Non-Overlap

One important constraint in the application is that a single worker cannot have overlapping availability time ranges. If a new row is inserted, or an existing row is updated that violates this constraint, a worker's availabilities should be merged in order to form one time range. The following is the trigger implementation:

```
create or replace function check_overlaps()
returns trigger as $$
declare temprow record;
begin
  for temprow in select starttime, endtime from availability where
workerid=new.workerid
      and (overlaps(starttime, endtime, new.starttime, new.endtime)
      or new.endtime=starttime or new.starttime=endtime)
      and (coalesce(old.starttime, new.starttime) <> starttime or
coalesce(old.endtime, new.endtime) <> endtime)
  loop
      raise notice 'OVERLAP S: %, E: %', temprow.starttime,
temprow.endtime;
      if temprow.starttime >= new.starttime and temprow.endtime <=
new.endtime then
            raise notice 'RESOLVING OVERLAP encompass strategy';
            -- if the conflict is fully encompassed by the new range
then delete the conflict
            delete from availability where workerid=new.workerid and
starttime=temprow.starttime and endtime=temprow.endtime;
      elsif temprow.starttime < new.starttime and temprow.endtime >
new.endtime then
            raise notice 'RESOLVING OVERLAP conflict strategy';
            -- if the conflict fully encompasses the new range then
ignore the new range
            -- can immediately return since there should be no other
conflicts
            return null;
      elsif temprow.starttime <= new.starttime and temprow.endtime <=
new.endtime then
            raise notice 'RESOLVING OVERLAP merge left strategy';
            delete from availability where workerid=new.workerid and
starttime=temprow.starttime and endtime=temprow.endtime;
            insert into availability (workerid, starttime, endtime)
values (new.workerid, temprow.starttime, new.endtime);
            return null;
```

```
      elsif temprow.endtime >= new.endtime and temprow.starttime <=
new.endtime then
            raise notice 'RESOLVING OVERLAP merge right strategy';
            new.endtime := temprow.endtime;
            delete from availability where workerid=new.workerid and
starttime=temprow.starttime and endtime=temprow.endtime;
      end if;
  end loop;
  return new;
end;
$$ language plpgsql;

create trigger trig_check_availability_overlap
before insert on availability
for each row
execute procedure check_overlaps();

create trigger trig_check_availability_overlap_upd
before update on availability
for each row
when (new.starttime < old.starttime or new.endtime > old.endtime)
execute procedure check_overlaps();
```

The trigger finds all conflicts with the new time range being added, and considers different resolution strategies. There are four main cases to note for this trigger: encompass, conflict, merge left, and merge right. Encompass occurs when a time range already in the database is fully encompassed by the new availability being added. Conflict happens when the conflicting time range already in the database fully encompasses the new availability being added. Merge left occurs when the overlap is on the left side of the new availability time range being added. Merge right occurs when the overlap is on the right side of the new availability time range being added. On insert, we always check for conflicts, but on update we only check for conflicts if the “size” of the time range was increased.

6.2 Account Covering Constraint

In our ER model, Account has an ISA-hierarchy with user, worker, and admin. We want to enforce the covering constraint, such that each account is at least of one type. The following is the trigger implementation:

```
create or replace function check_others_exist()
returns trigger as $$
declare count_users integer; count_workers integer; count_admins
integer; valid boolean := false;
begin
  select count(*) into count_users from users U where U.id = old.id;
```

```
  select count(*) into count_workers from workers W where W.id =
old.id;
  select count(*) into count_admins from admins A where A.id =
old.id;
  if TG_TABLE_NAME = 'users' and (count_workers=1 or count_admins=1)
then
    valid := true;
  end if;
  if TG_TABLE_NAME = 'workers' and (count_users=1 or count_admins=1)
then
    valid := true;
  end if;
  if TG_TABLE_NAME = 'admins' and (count_workers=1 or count_users=1)
then
    valid := true;
  end if;
  if valid=true then
    if TG_OP = 'UPDATE' then
      return new;
    elsif TG_OP = 'DELETE' then
      return old;
    else
      return null;
    end if;
  else
    raise notice 'ACCOUNT must belong to at least one type';
    return null;
  end if;
end;
$$ language plpgsql;

create or replace function check_at_least_one_account_type()
returns trigger as $$
declare count_users integer; count_workers integer; count_admins
integer;
begin
  select count(*) into count_users from users U where U.id = new.id;
  select count(*) into count_workers from workers W where W.id =
new.id;
  select count(*) into count_admins from admins A where A.id =
new.id;
  if count_users=1 or count_workers=1 or count_admins=1 then
    return null;
  else
```

```
    raise exception 'ACCOUNT must belong to at least one type';
  end if;
end;
$$ language plpgsql;

create trigger trig_check_others_exist
before update on users
for each row
when (NEW.id <> OLD.id)
execute procedure check_others_exist();

create trigger trig_check_others_exist
before update on workers
for each row
when (NEW.id <> OLD.id)
execute procedure check_others_exist();

create trigger trig_check_others_exist
before update on admins
for each row
when (NEW.id <> OLD.id)
execute procedure check_others_exist();

create trigger trig_check_others_exist_delete
before delete on users
for each row
execute procedure check_others_exist();

create trigger trig_check_others_exist_delete
before delete on workers
for each row
execute procedure check_others_exist();

create trigger trig_check_others_exist_delete
before delete on admins
for each row
execute procedure check_others_exist();

create constraint trigger trig_check_at_least_one_account_type
after insert or update on accounts
deferrable
for each row execute procedure check_at_least_one_account_type();
```

6.3 Non-Overlapping Bookings

We do not want any worker to be able to have bookings that overlap in their timing. If a booking is added or updated and overlaps with another booking the same worker has, we do not allow this change. The following is the trigger implementation:

```
create or replace function check_booking_overlaps()
returns trigger as $$
declare temprow record;
begin
  for temprow in select starttime, endtime from bookingdetails where
workerid=new.workerid
     and overlaps(starttime, endtime, new.starttime, new.endtime)
and bookingid <> new.bookingid
  loop
     raise notice 'OVERLAP S: %, E: %', temprow.starttime,
temprow.endtime;
     return null;
  end loop;
  return new;
end;
$$ language plpgsql;

drop trigger if exists trig_check_booking_overlap on bookingdetails;

create trigger trig_check_booking_overlap
before insert or update on bookingdetails
for each row
execute procedure check_booking_overlaps();
```

6.4 Referrer Check

One should not be allowed to refer themselves. As such, a constraint is necessary to prevent users from referring their own email. A check is done for every referral made that the email corresponding to the ID of the user making the referral is not the same as the email he/she is referring. The following is the trigger implementation:

```
create or replace function check_referrer()
returns trigger as $$
declare ref_email varchar(254);
begin
  ref_email := (select email from accounts where id=new.referrerid);
  if ref_email = new.email then
    raise notice 'Cannot refer yourself!';
    return null;
```

```
  else
    return new;
  end if;
end;
$$ language plpgsql;

drop trigger if exists check_referrer on refers;

create trigger check_referrer
before insert or update on refers
for each row
execute procedure check_referrer();
```

7. Interesting queries

7.1 Services Search Query

This query is designed to do a search on the services that are available in the application, as well as filter them by various parameters. Services are filterable by their name, price, category, service area, and rating. The query also joins the categories, and service area tables in order to get their names to display to the user from their IDs. The query only finds services that have availability in the future and only services that have not been deactivated by an administrator. This query filters the results depending on which search parameters are provided, and the complexity of it comes in when filtering the services by their average rating. Reviews are related to bookings, and bookings are related to services, so a left join on bookings and then on reviews is necessary to compute the average rating of each service. The base query is as follows:

```
select distinct S.serviceId, S.name as sname, description, price,
R.name as rname, C.name as cname, avg(V.rating)
from services S join cityregions R on R.regionid = S.regionid join
categories C on C.catid = S.catid
left join bookingdetails B on B.serviceid = S.serviceid and B.endtime
<= NOW()
left join reviews V on B.reviewid = V.reviewid
where exists (select 1 from availability A where A.workerid =
S.workerid and A.starttime > NOW())
group by s.serviceid, rname, cname
and not exists (select 1 from monitors M where M.serviceid =
S.serviceid and M.active = false)
```

If the search form is filled out with a service name as a query, for example, the following is added to the where clause:

```
and (S.name ILIKE $1 or S.description ILIKE $1)
```

$1 is replaced with whatever the user entered in the search box. Similar clauses are ANDed after this one (and before the `group by` clause) if the user specifies a maximum price, a category, or a service area that they want to filter by:

```
and price <= $n
```

```
and S.catid = $n
```

```
and S.regionid = $n
```

`$n` in this case is the number of the parameter in question, depending on how many parameters before it were put into the query. If the user selects a rating filter, the following is added after the `group by` clause:

```
having avg(V.rating) >= $n or avg(V.rating) is null
```

`$n` is again the number of the parameter in question, depending on how many parameters before it were put into the query.

7.2 Most Accepted Referrals Query

This query finds users who have the most accounts created as a result of their referral. Even if the user sends out thousands of referrals to random emails, the query will exclude any referrals that have not had an account created as a "result" of the referral (the person being referred enters their referrer's email on registration). It limits the results to the 3 top users, and excludes any accounts that have not had any accounts created as a result of their referral. The following is the implementation of the query:

```
select U.id, U.firstname, U.lastname, count(distinct R.email) from
accounts U
left join refers R on R.referrerid = U.id
where exists (select 1 from accounts A where A.email = R.email)
or not exists (select 1 from refers R1 where R1.referrerid = U.id)
group by U.id having count(distinct R.email) <> 0
limit 3
```

7.3 Worker Ratings Summary Queries

This group of queries first finds a worker's overall average rating based on reviews, also finds the average rating for each service category, and the average rating for each service area that the worker has worked in. The following is the implementation of the three queries:

```
select avg(R.rating) from reviews R join bookingdetails B on
B.reviewid=R.reviewid where B.workerid=$1
```

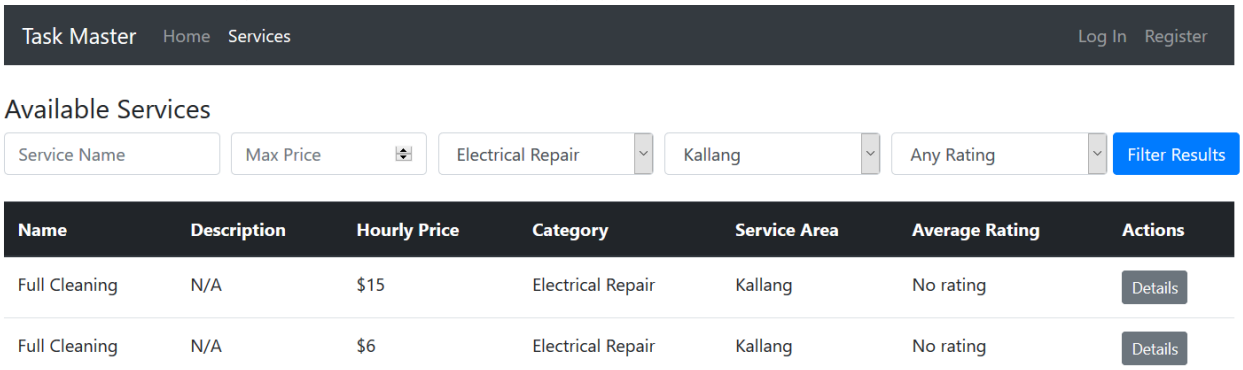
```
select C.catid, C.name, avg(R.rating) from reviews R join
bookingdetails B on B.reviewid = R.reviewid
```

```
  join services S on B.serviceid = S.serviceid join categories C on
S.catid = C.catid where B.workerid=$1
  group by C.catid
```

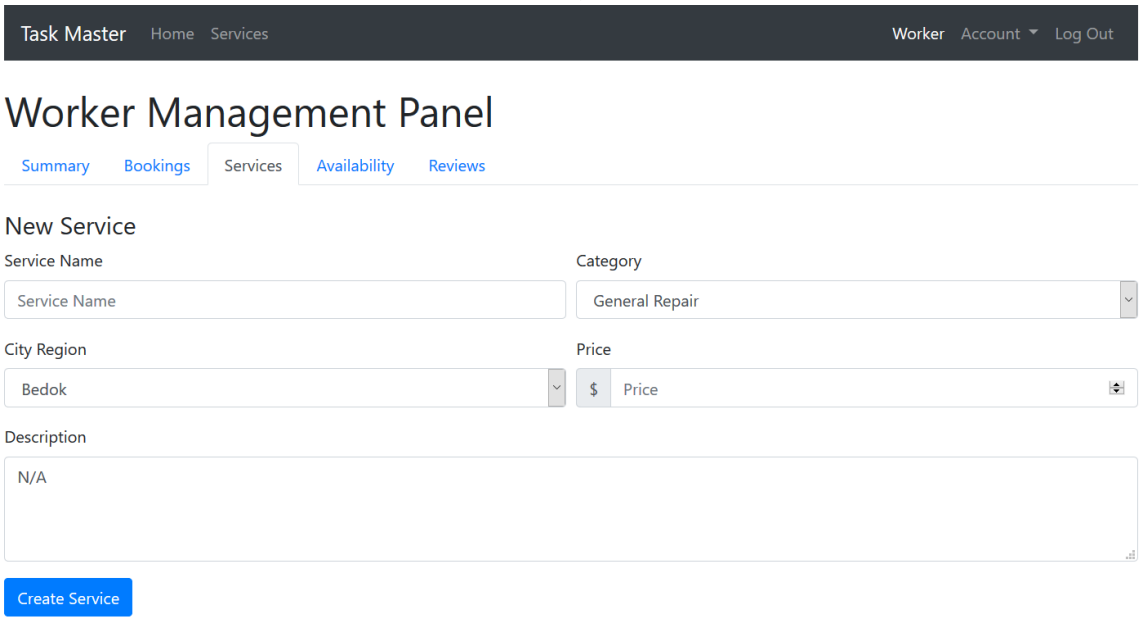
```
select CR.regionid, CR.name, avg(R.rating) from reviews R join
bookingdetails B on B.reviewid = R.reviewid
  join services S on B.serviceid = S.serviceid join cityregions CR on
S.regionid = CR.regionid where B.workerid=$1
  group by CR.regionid
```

8. Screenshots

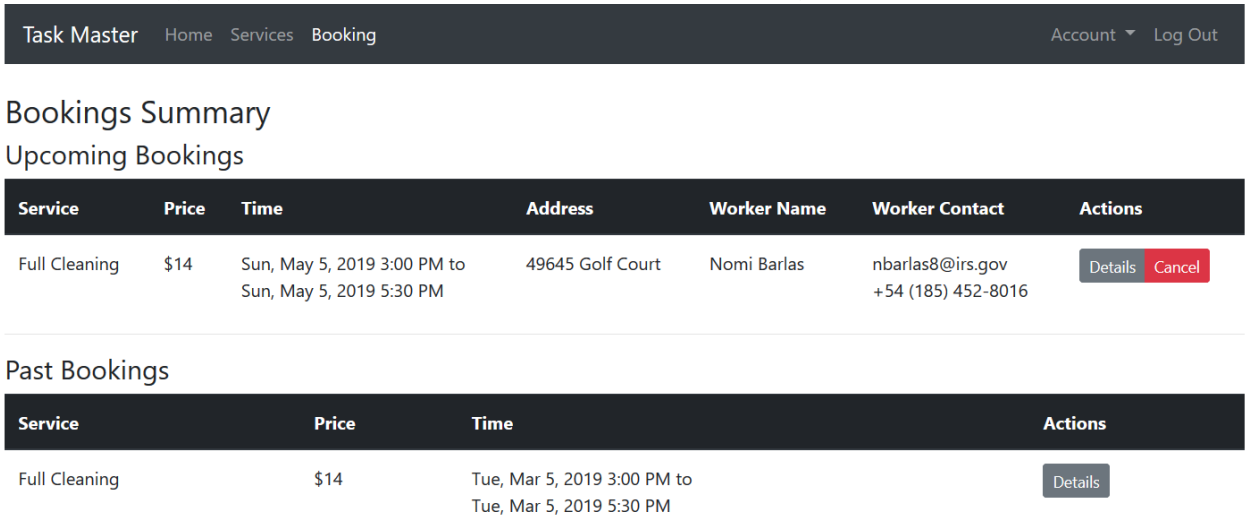
8.1 Services Page Filtered by Category and Service Area



8.2 Worker Panel: Add New Service Page



8.3 Booking Panel: User's Future and Past Bookings



9. Challenges and Lessons Learned

One of the biggest challenges our group encountered is a differing level of experience with the web development technologies we were using for the project. We spent a considerable amount of time helping each other be more comfortable with creating the website so that we could all make a fair contribution to the code. In particular, JavaScript was new to three out of the four group members, so a lot of learning had to be done on the go, as we implemented new features in the web application. Not having much JavaScript covered in class was also a challenge. Similarly, GitHub and Git were new to a portion of the group, so we also had to spend time on making sure everyone was able to complete their tasks successfully.

Another big challenge was designing and working with a data model that included so many relationship and entity sets. It was challenging to come up with a model that involved so many aspects to it for an application that seemed relatively simple to implement, but in the end it turned out that even with so many intricacies to the data model, we were still left feeling that we could implement additional functionality into the application if we wanted to make it comparable to existing applications that attempt to solve the same problem.

We felt it would be easier to work on the application if we met up together as a group for long development sessions, so that we could help each other in the case a problem arose, so we had to find times that worked for each of the group members. Assigning actionable tasks and monitoring development progress was another challenge we ran into. When a lot of the parts of the application depend on each other to fully work, it is difficult to split up the development in a way where one person's task does not interfere with another's. Defining these distinct actionable tasks was a challenge at the beginning of our project, but by the end it became easier, and each one of us was able to make more outside of our group meetings.

Completing this project helped us gain experience working with large data models, as well as dividing up the development that needed to be done in a way where one part did not depend on the other. In the end, even though we did not make the most perfect website, we were happy with what we were able to achieve in the timespan that we had to complete the project.