CS2102 Project
Report

Topic B: Crowdfunding

Group 23

Submitted by

Names of Students	Student ID
Chen Yinfang	A0179877Y
Li Kai	A0139021U
Peng Hanqiu	A0179035B
Tang Yifeng	A0179878X

Under the guidance of

Prof.Chan Chee Yong
&
Prof.Stéphane Bressan

School of Computing
National University of Singapore
SINGAPORE

Semester 2 2018

Abstract

Crowdfunding is the practice of funding a project or venture by raising many small amounts of money from a large number of people, typically via the Internet.

We present Givingchy, a practical crowdfunding platform focused on easy creation of donation projects. Givingchy has four major features: simple steps for creating projects; easy management of the latest progress and observation of every operation; focus on users' interested projects on different categories; users can comment on any project.

Givingchy is an excellent moderating crowdfunding platform that brings the various parties together to launch the idea. Its main body is a website that entrepreneurs can advertise their projects and users can browse and choose projects to fund. Everyone can be an entrepreneur and/or investor. Within the donation projects, members can not only donate but also comment on it. To facilitate the users searching, a search bar is built in which can enable the users to find their interested projects more conveniently. The database system keeps track and updates the information of all projects. Last but not least, several administrators have access to check, modify and delete projects and user accounts.

Contents

Chapter 1

Introduction

1.1 Design Concept

The main design concept behind the design of Givingchy is keeping the choice and donation procedure as easy as possible, make all of the information stored securely and stably enough, fulfill users' self-actualization in various areas.

Crowdfunding has been used to fund a wide range of for-profit ventures such as artistic and creative projects, medical expenses, travel, or community-oriented social entrepreneurship projects. Some requests, such as those to pay for optional expenses such as vacations, weddings, or cosmetic surgeries, are widely derided as internet begging or cyber-begging. So creating different categories such as philanthropy, technology, commerce and so on is essential.

1.2 Design Specifications

Database Server: PostgreSQL
Back-End Language: Go
Web Server: Nginx
Front-End Framework: VueJs

Chapter 2

Database

2.1 Entity-Relationship

2.1.1 Entity-Relationship Diagram

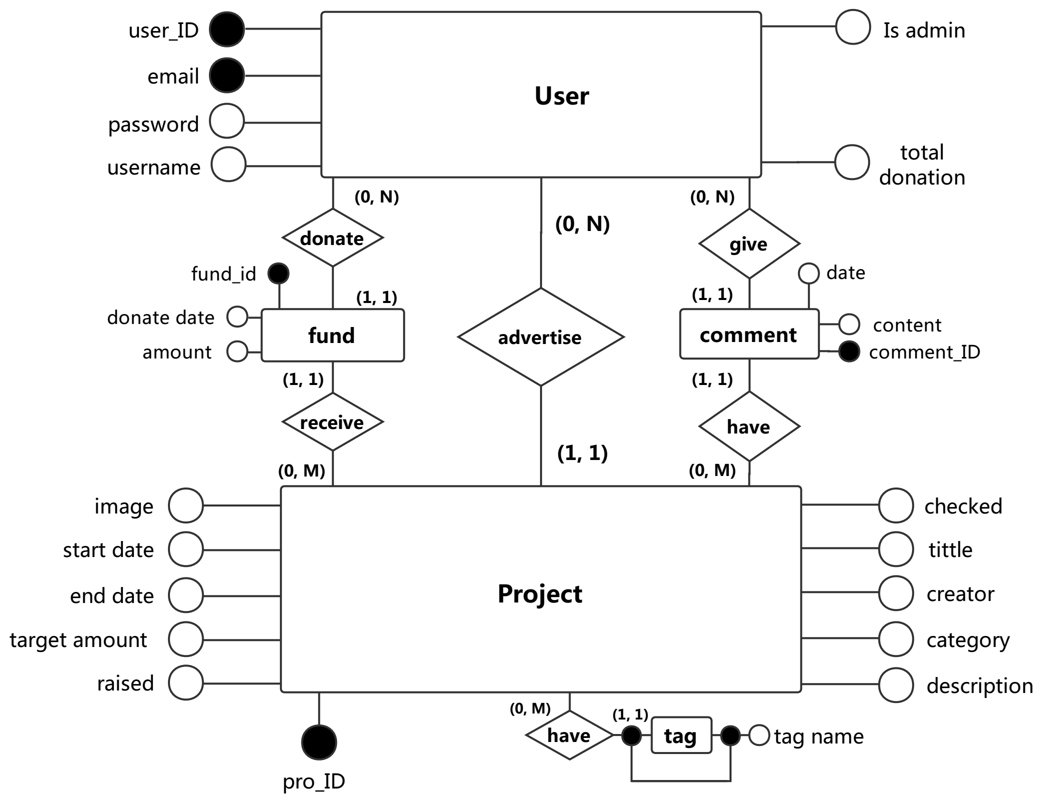


Figure 2.1: ER Diagram

2.1.2 Explanation

There are four entity sets in the ER diagram: User, Project, comment and fund.

User entity set is used to store the information of users. When a user signs up, he/she must provide a unique email address. Then the website will give him/her a unique user_ID which is the primary key of this entity. A user also need to set his usernames and passwords, and then he can sign in by his user_ID or email address. A user can be an administrator or not, which is indicated by the boolean attribute "Is admin". If "Is admin" is false, this user is a normal user and he cannot modify, delete any project (even his own project). On the contrary, if "Is admin" is true, this user can delete or modify the existing project. What's more, total donation records the total amount of fund that each user has donated. Besides, all the attributes in User entity set cannot be null.

A Project entity is identified by the primary key "pro_ID", which the system will allocate one after a user create a project. The "creator" attribute refers to the user_ID of its creator. Users should write the tittle, description, category, tag, start date and end date of this project, and target amount of fund, but the image is optional. The boolean attribute "checked" will be set to false by default which means that this project has not been checked by the administrator.

Advertise is a relationship set directly between User and Project. It means an account can advertise 0 project, 1 project or many projects. In addition, the cardinality of (0, n) and (1, 1) in the diagram indicates that a project can only be created by one account.

Another two entity sets are fund and comment, they are used when there is a large amount of record describing funds (transactions) and comments, an account donates many times to one specific project, as well as a project has many comments from the same user. An account can donate zero or many times to the project and a project can receive zero or many times of fund. When a donation succeeds, the Project will add the amount of this fund to its "raised" value. The donation date is also stored in the "donate date". Comment entity will record the words and on which day the users say. As for the tag, a project may have several its own tags so it is a weak entity whose primary key should be decided by both the project's id and the tag name.

Based on the above ER diagram, we have created the relational schema for our database.

2.2 Highlight

This project is composed of three layers: front-end layer, back-end layer and database layer. Each layer is transparent to each other by only providing several APIs. This design of structure is able to reduce the coupling of the whole project and make programmers more easier to write, update or even reconstruct the project.

In order to achieve this goal, for example, we avoid using any direct SQL query in the back-end layer but using procedure storage especially functions and triggers. In total, we use **34 functions and 8 triggers** among all tables. These functions and triggers restrict what others without direct access to the database can do.

However, the database is still possibly ruined by bad or careless users. So we design another extra table named **Logs table** to record each operation on database. We use both triggers and functions to do it automatically. If something wrong happens, administrators are able to read the logs to find out the reason.

On the other hand, the database could be extremely large which requires high performance when user is doing a query. Since we limit the number of tuples users can get, it's not necessary to query the whole table each time. We use **cursor** for every query to reduce the cost.

2.3 Relational Schema

We have used many triggers and functions so, to make it clearer, here we just show DDL, one complex function and one complex trigger. We have attached all DML to the APPENDIX.

```
--DDL--
--extension used to ignore low cases and caps
create extension if not exists citext;
--extension used to encrypt the function
create extension if not exists pgcrypto;

drop table if exists users cascade;
drop table if exists categories cascade;
drop table if exists projects cascade;
drop table if exists payments cascade;
drop table if exists comments cascade;
drop table if exists logs cascade;
drop table if exists tags cascade;
```

```sql
--create user table
create table if not exists users (
    user_id serial primary key,
    email citext unique not null,
    password varchar(255) not null,
    username citext not null,
    total_donation numeric(10, 2) not null default 0,
    image citext,
    is_admin boolean not null default false
);
--create category table to classify the project
create table if not exists categories (
    name citext unique primary key,
    proj_num integer not null default 0 check (proj_num >= 0)
);
--create project table
create table if not exists projects (
    project_id serial primary key,
    title varchar(100) not null,
    user_id integer not null references users (user_id) on update
        cascade on delete cascade,
    category citext not null references categories (name) on update
        cascade on delete cascade,
    description text not null,
    verified boolean not null default false,
    image citext,
    amount_raised numeric(10, 2) not null check (amount_raised >=
        0) default 0,
    amount_required numeric(10, 2) not null check (amount_required
        > 0), --10 sf, 2dp
    start_time timestamp not null default now(),
    end_time timestamp not null check (start_time <= end_time)
);
--create tag table and the difference between 'tags' and 'category' is that
--'tag' is just like a small category which user add to their projects
--themselves but 'category' is set by admin by default
create table if not exists tags(
    project_id integer not null references projects (project_id) on
        update cascade on delete cascade,
    tag_name citext not null,
    primary key(project_id, tag_name)
);
--create payment table to record users' donation
create table if not exists payments (
    id serial primary key,
    user_id integer not null references users (user_id) on update
        cascade on delete cascade,
    project_id integer not null references projects (project_id) on
        update cascade on delete cascade,
    moment timestamp not null default now(),
    amount numeric(10, 2) not null check (amount > 0) --10 sf, 2dp
);
--create comment table to record the comments on specific project
create table if not exists comments (
    id serial primary key,
    user_id integer not null references users (user_id) on update
        cascade on delete cascade,
    project_id integer not null references projects (project_id) on
        update cascade on delete cascade,
    moment timestamp not null default now(),
    content text not null
);
--create log table to record the each operation users make
create table if not exists logs (
    id serial primary key,
    user_id integer default null,
    project_id integer default null,
    moment timestamp not null default now(),
    content text not null,
    log_level integer not null
);
--DML--
--This function is support for searching a project
-- search projects by one keyword
create or replace function search_project(_keyword citext,
    _num_per_page int, _idx_page int)
returns setof project_row as $$
declare
    proj project_row%rowtype;
    proj_row_cursor refcursor;
    i int;
begin
```

```
    insert into logs(content, log_level)
        values ('Search␣projects', 1);
    -- open a cursor to avoid doing a query among the whole table
    open proj_row_cursor for
        select *
        from projects
        where title like '%' || _keyword || '%';
    -- pattern "'%' || _keyword || '%'" equals to
    -- regex [a-z||A-Z]* _keyword [a-z||A-Z]*
    -- which means there exist more than or equal to zero
    -- words on both sides of _keyword
    move absolute (_idx_page - 1) * _num_per_page from
        proj_row_cursor;
    i := 0;
    loop
        if i >= _num_per_page then
            exit;
        end if;
        i := i + 1;
        -- get the current tuple
        fetch proj_row_cursor into proj;
        exit when not found;
        return next proj;
    end loop;
    close proj_row_cursor;
    return;
end
$$ language plpgsql;
--This trigger is support for donating
create trigger take_log after insert or update or delete on payments
for each row execute procedure create_log_user_proj('␣on␣payments');
-- trigger for any payments
-- if delete a payment, we should decrease corresponding user's and
    project's donation
-- update a payment, we should change (or decrease and then increase)
-- insert a payment, we should increase
create or replace function donate_trigger()
returns trigger as $$
begin
    if (tg_op = 'DELETE') then
        -- old is the original tuple whom we are going to operates on
        update users
            set total_donation = total_donation − old.amount
            where user_id = old.user_id;
        update projects
            set amount_raised = amount_raised − old.amount
            where project_id = old.project_id;
        return old;
    elsif (tg_op = 'UPDATE') then
        update users
            set total_donation = total_donation − old.amount
            where user_id = old.user_id;
        update projects
            set amount_raised = amount_raised − old.amount
            where project_id = old.project_id;
        −− new is the tuple which is added just now
        update users
            set total_donation = total_donation + new.amount
            where user_id = new.user_id;
        update projects
            set amount_raised = amount_raised + new.amount
            where project_id = new.project_id;
        return new;
    elsif (tg_op = 'INSERT') then
        update users
            set total_donation = total_donation + new.amount
            where user_id = new.user_id;
        update projects
            set amount_raised = amount_raised + new.amount
            where project_id = new.project_id;
        return new;
    end if;
end;
$$ language plpgsql;
```

Chapter 3

Web Page Design

3.1 Login and Sign-up Page

This login page requires users to enter email address and password . . . . . in order to access the main page of our website. New user is able to create new account by clicking “click here to register” button which will be directed to a sign up page.

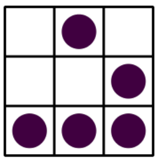
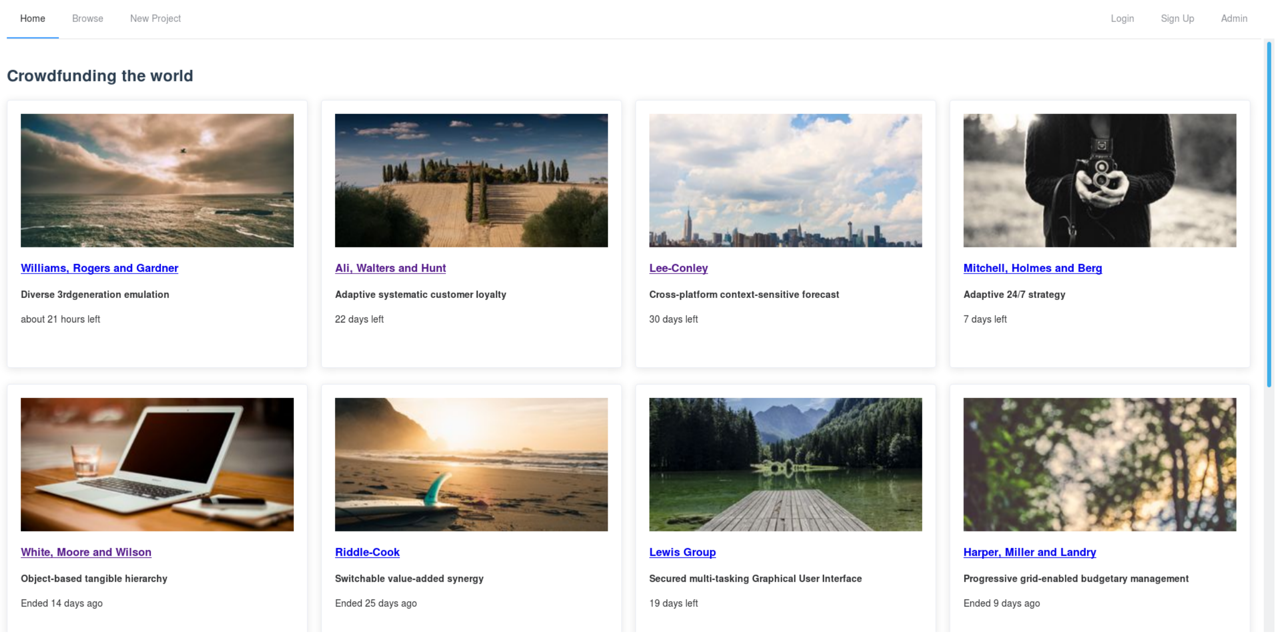


Figure 3.1: Login page

This sign up page requires users to fill up ((((((full name, email address and password))) with certain requirements. Email address must follow the format of XX@XX.com. Password must be at least 8 characters with at least one number, one alphabet and one alphabet in upper case. Invalid entries will not pass this page and warnings will be shown.

3.2 Project Page

3.2.1 Home Page



Crowdfunding the world

Williams, Rogers and Gardner
Diverse 3rdgeneration emulation
about 21 hours left

Ali, Walters and Hunt
Adaptive systematic customer loyalty
22 days left

Lee-Conley
Cross-platform context-sensitive forecast
30 days left

Mitchell, Holmes and Berg
Adaptive 24/7 strategy
7 days left

White, Moore and Wilson
Object-based tangible hierarchy
Ended 14 days ago

Riddle-Cook
Switchable value-added synergy
Ended 25 days ago

Lewis Group
Secured multi-tasking Graphical User Interface
19 days left

Harper, Miller and Landry
Progressive grid-enabled budgetary management
Ended 9 days ago

Figure 3.2: Home page of Givingchy

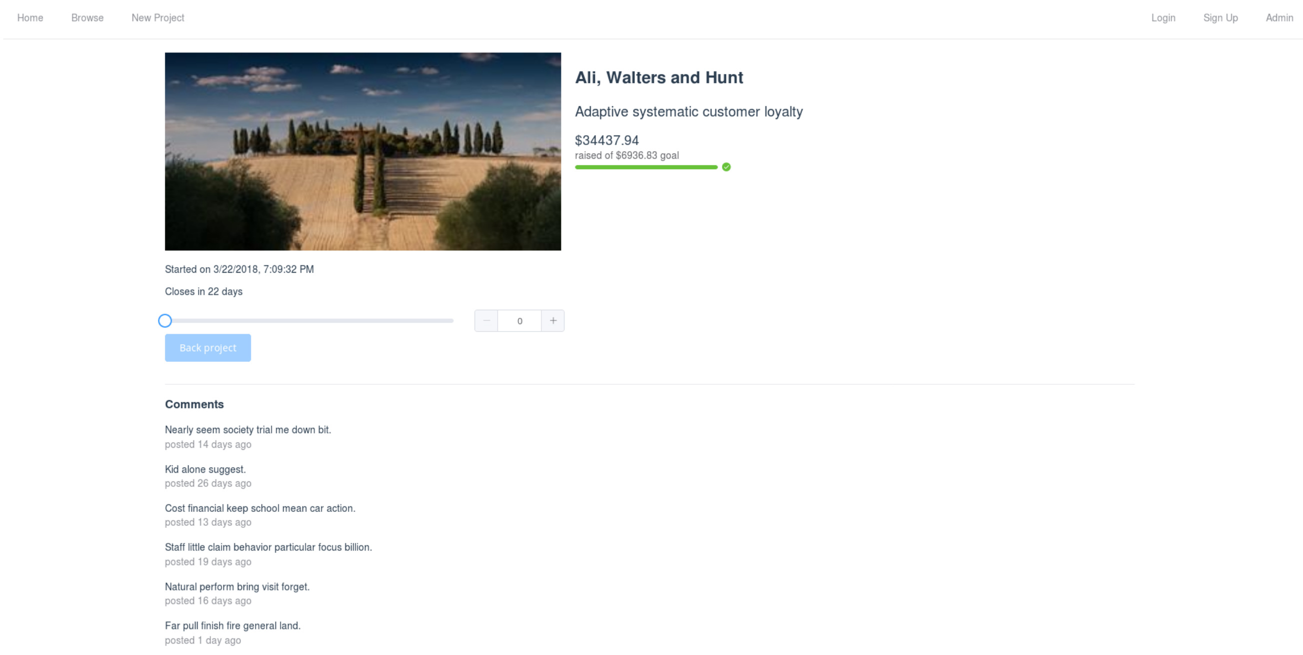
3.2.2 Project Profile Page

Browsing and Donating Project Page

This is the profile page of a project with a progress bar showing the current funding progress. Once the progress bar hits 100%, the project is funded. The status of the project will be changed to “completed” from “not completed”. Users can still view projects that have already completed the targets. Administrators will delete funded projects on the monthly basis. Users can enter in the amount field at “donation amount” and click “donate” to invest in the project.

Creating Project Page

This is the page for creating a project by any user. Users can fill up the form and click “create” to submit information to our database.



Ali, Walters and Hunt

Adaptive systematic customer loyalty

$34437.94
raised of $6936.83 goal

Started on 3/22/2018, 7:09:32 PM

Closes in 22 days

Comments

Nearly seem society trial me down bit.
posted 14 days ago

Kid alone suggest.
posted 26 days ago

Cost financial keep school mean car action.
posted 13 days ago

Staff little claim behavior particular focus billion.
posted 19 days ago

Natural perform bring visit forget.
posted 16 days ago

Far pull finish fire general land.
posted 1 day ago

Figure 3.3: Browsing a project of Givingchy

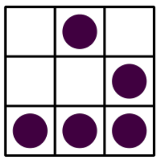


Figure 3.4: Donating a project of Givingchy

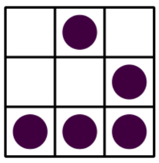


Figure 3.5: Creating one project by a user

3.3 User Profile Page

3.3.1 Non-administrator Profile Page

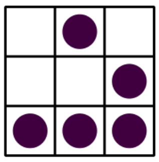


Figure 3.6: Non-administrator profile page

3.3.2 Administrator Profile Page

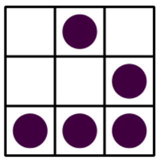


Figure 3.7: Administrator profile page

Chapter 4

APPENDIX

```
--DML--
--Log--
--To record every operation happening in the database--
drop type if exists log_row cascade;

-- create type for collecting results from query
create type log_row as (
    id int,
    user_id int,
    project_id int,
    moment timestamp,
    content text,
    log_level int
);

create or replace function all_logs()
returns setof log_row as $$
declare
    lrow log_row%rowtype;
begin
    for lrow in
        select *
        from logs
    loop
        return next lrow;
    end loop;
    return;
end
```

```
$$ language plpgsql;

create or replace function all_user_logs(_user_id int)
returns setof log_row as $$
declare
    lrow log_row%rowtype;
begin
    for lrow in
        select *
        from logs
        where user_id = _user_id
    loop
        return next lrow;
    end loop;
    return;
end
$$ language plpgsql;

create or replace function all_project_logs(_project_id int)
returns setof log_row as $$
declare
    lrow log_row%rowtype;
begin
    for lrow in
        select *
        from logs
        where project_id = _project_id
    loop
        return next lrow;
    end loop;
    return;
end
$$ language plpgsql;

-- triggers for those tables who only have user_id
create or replace function create_log_user()
returns trigger as $$
begin
    if (tg_op = 'DELETE') then
        insert into logs(user_id, content, log_level)
            values (old.user_id, 'Delete op' || tg_argv[0], 0);
        return old;
    elsif (tg_op = 'UPDATE') then
        insert into logs(user_id, content, log_level)
            values (new.user_id, 'Update␣op' || tg_argv[0], 0);
        return new;
    elsif (tg_op = 'INSERT') then
        insert into logs(user_id, content, log_level)
            values (new.user_id, 'Insert␣op' || tg_argv[0], 0);
        return new;
    end if;
    return null;
end
$$ language plpgsql;

-- triggers for those tables who only have proj_id
create or replace function create_log_proj()
returns trigger as $$
begin
    if (tg_op = 'DELETE') then
        insert into logs(project_id, content, log_level)
            values (old.project_id, 'Delete␣op' || tg_argv[0], 0);
        return old;
    elsif (tg_op = 'UPDATE') then
        insert into logs(project_id, content, log_level)
            values (new.project_id, 'Update␣op' || tg_argv[0], 0);
        return new;
    elsif (tg_op = 'INSERT') then
        insert into logs(project_id, content, log_level)
            values (new.project_id, 'Insert␣op' || tg_argv[0], 0);
        return new;
    end if;
    return null;
end
$$ language plpgsql;

-- triggers for those tables who have both user_id and proj_id
create or replace function create_log_user_proj()
returns trigger as $$
begin
    if (tg_op = 'DELETE') then
        insert into logs(user_id, project_id, content, log_level)
```

```
            values (old.user_id, old.project_id, 'Delete␣op' || tg_argv[0],
                0);
        return old;
    elsif (tg_op = 'UPDATE') then
        insert into logs(user_id, project_id, content, log_level)
            values (new.user_id, new.project_id, 'Update␣op' || tg_argv
                [0], 0);
        return new;
    elsif (tg_op = 'INSERT') then
        insert into logs(user_id, project_id, content, log_level)
            values (new.user_id, new.project_id, 'Insert␣op' || tg_argv
                [0], 0);
        return new;
    end if;
    return null;
end
$$ language plpgsql;

-- triggers for those tables who has no any id
create or replace function create_log()
returns trigger as $$
begin
    if (tg_op = 'DELETE') then
        insert into logs(content, log_level)
            values ('Delete␣op' || tg_argv[0], 0);
        return old;
    elsif (tg_op = 'UPDATE') then
        insert into logs(content, log_level)
            values ('Update␣op' || tg_argv[0], 0);
        return new;
    elsif (tg_op = 'INSERT') then
        insert into logs(content, log_level)
            values ('Insert␣op' || tg_argv[0], 0);
        return new;
    end if;
    return null;
end
$$ language plpgsql;


--Category--
```

```
drop type if exists cate_row cascade;
drop trigger if exists take_log on categories;

create type cate_row as (
    name citext,
    proj_num int
);

-- select all categories
create or replace function all_categories()
returns setof cate_row as $$
declare
    cate cate_row%rowtype;
begin
    insert into logs(content, log_level)
        values ('Select all categories', 1);
    for cate in
        select *
        from categories
    loop
        return next cate;
    end loop;
    return;
end
$$ language plpgsql;

-- create a category
create or replace function create_categories(_name citext)
returns void as $$
    insert into categories(name)
        values(_name);
$$ language sql;

-- create trigger for taking log
create trigger take_log after insert or update or delete on categories
for each row execute procedure create_log(' on categories');


--Comments--
drop type if exists comment_row cascade;
drop trigger if exists take_log on comments;
```

```
create type comment_row as (
    id int,
    user_id int,
    project_id int,
    moment timestamp,
    content text
);

-- select all comments
create or replace function all_comments(_num_per_page int,
    _idx_page int)
returns setof comment_row as $$
declare
    comm comment_row%rowtype;
    comm_row_cursor refcursor;
    i int;
begin
    insert into logs(content, log_level)
        values ('Select all comments', 1);
    -- open a cursor to avoid doing a query among the whole table
    open comm_row_cursor for
        select *
        from comments;
    -- move the cursor to the very place we need
    move absolute (_idx_page - 1) * _num_per_page from
        comm_row_cursor;
    i := 0;
    loop
        if i >= _num_per_page then
            exit;
        end if;
        i := i + 1;
        fetch comm_row_cursor into comm;
        exit when not found;
        return next comm;
    end loop;
    close comm_row_cursor;
    return;
end
$$ language plpgsql;
```

```
-- select one project's comments
create or replace function all_project_comments(_project_id int,
    _num_per_page int, _idx_page int)
returns setof comment_row as $$
declare
    comm comment_row%rowtype;
    comm_row_cursor refcursor;
    i int;
begin
    insert into logs(project_id, content, log_level)
        values (_project_id, 'Select project's comments', 1);
    -- open a cursor to avoid doing a query among the whole table
    open comm_row_cursor for
        select *
        from comments
        where project_id = _project_id;
    -- move the cursor to the very place we need
    move absolute (_idx_page - 1) * _num_per_page from
       comm_row_cursor;
    i := 0;
    loop
        if i >= _num_per_page then
            exit;
        end if;
        i := i + 1;
        fetch comm_row_cursor into comm;
        exit when not found;
        return next comm;
    end loop;
    close comm_row_cursor;
    return;
end
$$ language plpgsql;

-- select one user's comments
create or replace function all_user_comments(_user_id int,
    _num_per_page int, _idx_page int)
returns setof comment_row as $$
declare
    comm comment_row%rowtype;
    comm_row_cursor refcursor;
    i int;
begin
    insert into logs(user_id, content, log_level)
        values (_user_id, 'Select␣user_s␣comments', 1);
    -- open a cursor to avoid doing a query among the whole table
    open comm_row_cursor for
        select *
        from comments
        where user_id = _user_id;
    -- move the cursor to the very place we need
    move absolute (_idx_page - 1) * _num_per_page from
        comm_row_cursor;
    i := 0;
    loop
        if i >= _num_per_page then
            exit;
        end if;
        i := i + 1;
        fetch comm_row_cursor into comm;
        exit when not found;
        return next comm;
    end loop;
    close comm_row_cursor;
    return;
end
$$ language plpgsql;

create or replace function create_comment(_user_id int, _project_id
    int, _content text)
returns integer as $$
    insert into comments(user_id, project_id, content)
        values(_user_id, _project_id, _content);
    select max(id)
        from comments;
$$ language sql;

create or replace function update_comment(_comment_id int, _content
    text)
returns void as $$
    update comments
```

```
        set content = _content
        where id = _comment_id;
$$ language sql;

create or replace function delete_comment(_comment_id int)
returns void as $$
    delete from comments
        where id = _comment_id;
$$ language sql;

-- create trigger for taking log
create trigger take_log after insert or update or delete on comments
for each row execute procedure create_log_user_proj('␣on␣comments');


--Donation--
drop type if exists payments_row cascade;
drop trigger if exists take_log on payments;
drop trigger if exists donate on payments;

create type payments_row as (
    id int,
    user_id int,
    project_id int,
    moment timestamp,
    amount numeric
);

-- select all payments
create or replace function all_payments(_num_per_page int,
    _idx_page int)
returns setof payments_row as $$
declare
    pay payments_row%rowtype;
    pay_row_cursor refcursor;
    i int;
begin
    insert into logs(content, log_level)
        values ('Select␣all␣payments', 1);
    -- open a cursor to avoid doing a query among the whole table
    open pay_row_cursor for
```

```
        select *
        from payments;
    -- move the cursor to the very place we need
    move absolute (_idx_page - 1) * _num_per_page from
        pay_row_cursor;
    i := 0;
    loop
        if i >= _num_per_page then
            exit;
        end if;
        i := i + 1;
        fetch pay_row_cursor into pay;
        exit when not found;
        return next pay;
    end loop;
    close pay_row_cursor;
    return;
end
$$ language plpgsql;

-- select one project's payments
create or replace function all_project_payments(_project_id int,
    _num_per_page int, _idx_page int)
returns setof payments_row as $$
declare
    pay payments_row%rowtype;
    pay_row_cursor refcursor;
    i int;
begin
    insert into logs(project_id, content, log_level)
        values (_project_id, 'Select␣project␣s␣payments', 1);
    -- open a cursor to avoid doing a query among the whole table
    open pay_row_cursor for
        select *
        from payments
        where project_id = _project_id;
    -- move the cursor to the very place we need
    move absolute (_idx_page - 1) * _num_per_page from
        pay_row_cursor;
    i := 0;
    loop
```

```
        if i >= _num_per_page then
            exit;
        end if;
        i := i + 1;
        fetch pay_row_cursor into pay;
        exit when not found;
        return next pay;
    end loop;
    close pay_row_cursor;
    return;
end
$$ language plpgsql;

-- select one usr's payments
create or replace function all_user_payments(_user_id int,
    _num_per_page int, _idx_page int)
returns setof payments_row as $$
declare
    pay payments_row%rowtype;
    pay_row_cursor refcursor;
    i int;
begin
    insert into logs(user_id, content, log_level)
        values (_user_id, 'Select␣all␣payments', 1);
    -- open a cursor to avoid doing a query among the whole table
    open pay_row_cursor for
        select *
        from payments
        where user_id = _user_id;
    -- move the cursor to the very place we need
    move absolute (_idx_page - 1) * _num_per_page from
        pay_row_cursor;
    i := 0;
    loop
        if i >= _num_per_page then
            exit;
        end if;
        i := i + 1;
        fetch pay_row_cursor into pay;
        exit when not found;
        return next pay;
    end loop;
    close pay_row_cursor;
    return;
end
$$ language plpgsql;

create or replace function create_payment(_user_id int, _project_id
    int, _amount numeric)
returns integer as $$
    insert into payments(user_id, project_id, amount)
        values(_user_id, _project_id, _amount);
    select max(id)
        from payments;
$$ language sql;

create or replace function update_payment(_payment_id int, _amount
    numeric)
returns void as $$
    update payments
        set amount = _amount
        where id = _payment_id;
$$ language sql;

create or replace function delete_payment(_payment_id int)
returns void as $$
    delete from payments
        where id = _payment_id;
$$ language sql;

-- create trigger for donating
create trigger take_log after insert or update or delete on payments
for each row execute procedure create_log_user_proj('␣on␣payments');

-- trigger for any payments
-- if delete a payment, we should decrease corresponding user's and
    project's donation
-- update a payment, we should change (or decrease and then increase)
-- insert a payment, we should increase
create or replace function donate_trigger()
returns trigger as $$
begin
```

```
    if (tg_op = 'DELETE') then
        -- old is the original tuple whom we are going to operates on
        update users
            set total_donation = total_donation - old.amount
            where user_id = old.user_id;
        update projects
            set amount_raised = amount_raised - old.amount
            where project_id = old.project_id;
        return old;
    elsif (tg_op = 'UPDATE') then
        update users
            set total_donation = total_donation - old.amount
            where user_id = old.user_id;
        update projects
            set amount_raised = amount_raised - old.amount
            where project_id = old.project_id;
        -- new is the tuple which is added just now
        update users
            set total_donation = total_donation + new.amount
            where user_id = new.user_id;
        update projects
            set amount_raised = amount_raised + new.amount
            where project_id = new.project_id;
        return new;
    elsif (tg_op = 'INSERT') then
        update users
            set total_donation = total_donation + new.amount
            where user_id = new.user_id;
        update projects
            set amount_raised = amount_raised + new.amount
            where project_id = new.project_id;
        return new;
    end if;
end;
$$ language plpgsql;

-- create trigger for taking log
create trigger donate after insert or update or delete on payments
for each row execute procedure donate_trigger();
```

```
−−Projects−−
drop type if exists project_row cascade;
drop trigger if exists take_log on projects;
drop trigger if exists update_cate on projects;

create type project_row as (
    project_id int,
    title varchar(100),
    user_id int,
    category citext,
    description text,
    verified boolean,
    image citext,
    amount_raised numeric,
    amount_required numeric,
    start_time timestamp,
    end_time timestamp
);

−− select all projects
create or replace function all_projects(_num_per_page int, _idx_page
    int)
returns setof project_row as $$
declare
    proj project_row%rowtype;
    proj_row_cursor refcursor;
    i int;
begin
    insert into logs(content, log_level)
        values ('Select␣all␣projects', 1);
    open proj_row_cursor for
        select ∗
        from projects;
    move absolute (_idx_page − 1) ∗ _num_per_page from
        proj_row_cursor;
    i := 0;
    loop
        if i >= _num_per_page then
            exit;
        end if;
        i := i + 1;
        fetch proj_row_cursor into proj;
        exit when not found;
        return next proj;
    end loop;
    close proj_row_cursor;
    return;
end
$$ language plpgsql;

-- search projects by one keyword
create or replace function search_project(_keyword citext,
    _num_per_page int, _idx_page int)
returns setof project_row as $$
declare
    proj project_row%rowtype;
    proj_row_cursor refcursor;
    i int;
begin
    insert into logs(content, log_level)
        values ('Search␣projects', 1);
    -- open a cursor to avoid doing a query among the whole table
    open proj_row_cursor for
        select *
        from projects
        where title like '%' || _keyword || '%';
    -- pattern "'%' || _keyword || '%'" equals to
    -- regex [a-z||A-Z]+ _keyword [a-z||A-Z]+
    -- which means there exist more than or equals to zero
    -- words on both sides of _keyword
    move absolute (_idx_page - 1) * _num_per_page from
        proj_row_cursor;
    i := 0;
    loop
        if i >= _num_per_page then
            exit;
        end if;
        i := i + 1;
        -- get the current tuple
        fetch proj_row_cursor into proj;
        exit when not found;
        return next proj;
    end loop;
    close proj_row_cursor;
    return;
end
$$ language plpgsql;


create or replace function create_project(
    _title varchar(100),
    _user_id int,
    _category citext,
    _description text,
    _image citext,
    _amount_required numeric,
    _end_time timestamp)
returns integer as $$
    insert into projects (title, user_id, category, description, image,
        amount_required, end_time)
        values (_title, _user_id, _category, _description, _image,
            _amount_required, _end_time);
    select max(project_id)
        from projects
$$ language sql;

create or replace function get_project(_project_id int)
returns project_row as $$
declare
    proj_row project_row%rowtype;
begin
    insert into logs(project_id, content, log_level)
        values (_project_id, 'Select␣project', 1);
    select *
        from projects
        into proj_row
        where project_id = _project_id;
    return proj_row;
end
$$ language plpgsql;

-- trigger for any projects
-- if delete a project, we should decrease corresponding category's number
   of projects
-- update a project, we should change (or decrease and then increase)
-- insert a project, we should increase
create or replace function update_cate_num()
returns trigger as $$
begin
   if (tg_op = 'DELETE') then
      update categories
         set proj_num = proj_num - 1
         where name = old.category;
      return old;
   elsif (tg_op = 'UPDATE') then
      update categories
         set proj_num = proj_num - 1
         where name = old.category;
      update categories
         set proj_num = proj_num + 1
         where name = new.category;
      return new;
   elsif (tg_op = 'INSERT') then
      update categories
         set proj_num = proj_num + 1
         where name = new.category;
      return new;
   end if;
   return null;
end
$$ language plpgsql;

create trigger update_cate after insert or delete on projects
for each row execute procedure update_cate_num();

create trigger take_log after insert or update or delete on projects
for each row execute procedure create_log_user_proj('␣on␣projects');


--Tags--
drop type if exists tag_row cascade;
drop trigger if exists take_log on tags;
```

```
create type tag_row as (
    project_id int,
    tag_name citext
);

create or replace function all_tags()
returns setof citext as $$
declare
    _tag_name citext;
begin
    insert into logs(content, log_level)
        values ('Select␣all␣tags', 1);
    for _tag_name in
        select distinct tag_name
        from tags
    loop
        return next _tag_name;
    end loop;
    return;
end
$$ language plpgsql;

create or replace function create_tag(
    _project_id int,
    _tag_name citext)
returns citext as $$
    insert into tags (project_id, tag_name)
        values (_project_id, _tag_name);
    select _tag_name;
$$ language sql;

create or replace function get_project_s_tags(
    _project_id int)
returns setof citext as $$
declare
    _tag_name citext;
begin
    insert into logs(project_id, content, log_level)
        values (_project_id, 'Select␣project_s␣tags', 1);
    for _tag_name in
        select tag_name
        from tags
        where project_id = _project_id
    loop
        return next _tag_name;
    end loop;
    return;
end
$$ language plpgsql;

create or replace function get_tag_s_projects(
    _tag_name citext)
returns setof int as $$
declare
    _project_id int;
begin
    insert into logs(content, log_level)
        values ('Select␣tag_s␣projects', 1);
    for _project_id in
        select project_id
        from tags
        where tag_name = _tag_name
    loop
        return next _project_id;
    end loop;
    return;
end
$$ language plpgsql;

create trigger take_log after insert or update or delete on tags
for each row execute procedure create_log_proj('␣on␣tags');
```