CS2102
Database Systems

Team 18

Topic B: Crowdfunding

Deric Khoo Jia Wei	*A0155389N*
Fong Wei Zheng	*A0156708X*
Kion Shi Rong	*A0149782N*
Chen Zhi Liang	*A0141010L*

Introduction

The crowdfunding website provides the following functionalities:

- *Support creation/deletion/update of data*:
  - Users can create crowdfunding projects.
  - Users can choose to delete their own projects.
  - Users can edit the details of their own projects – including project description, dates and other important information.
  - Users can edit their personal profile information.
  - Admin users can perform the above functionalities to any projects.

- *Support browsing and searching of data*:
  - Users can use any relevant attributes to search for a crowdfunding project.

- *Support retrieval of interesting statistics on data*:
  - Projects display the accumulated amount of investments invested into them.
  - Users can check the accumulated amount of investment he or she has invested since the creation of his or her account
  - Users can see how many of his or her project is inactive or has passed the stipulated end date.

In our crowdfunding website, we have two self-explanatory entities – *Users* and *Projects*. These two entities are related in relationships called *Investment* because users can choose to invest in projects of their choice. In the ER diagrams in the following section, we can see the various attributes and keys associated with each entity and relationships.

Project Specifications

Stack Infrastructure: *WAPP Bitnami*

Frontend: *HTML, CSS*

Backend: *PHP*

Database: *SQL*

Entity Relationship Diagrams

Fig 1. User Entity

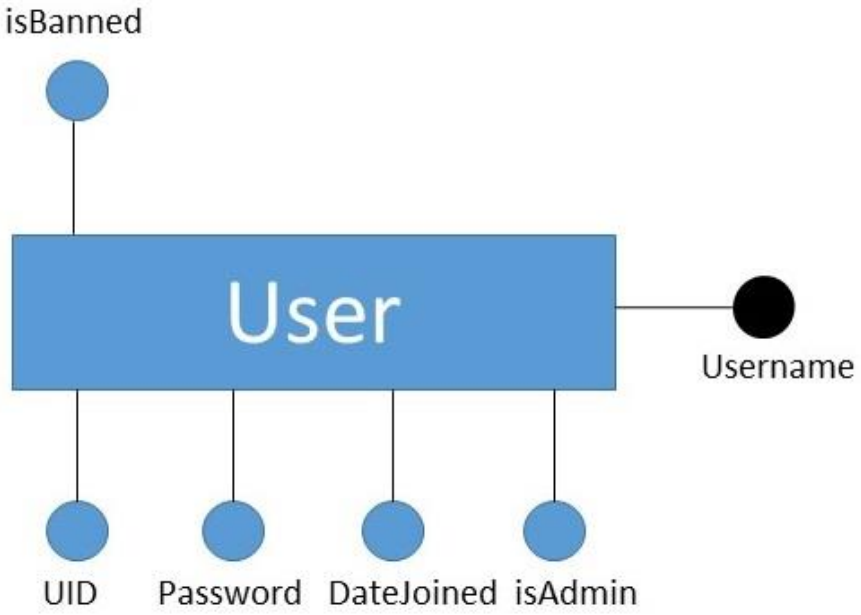


Fig 2. Project Entity

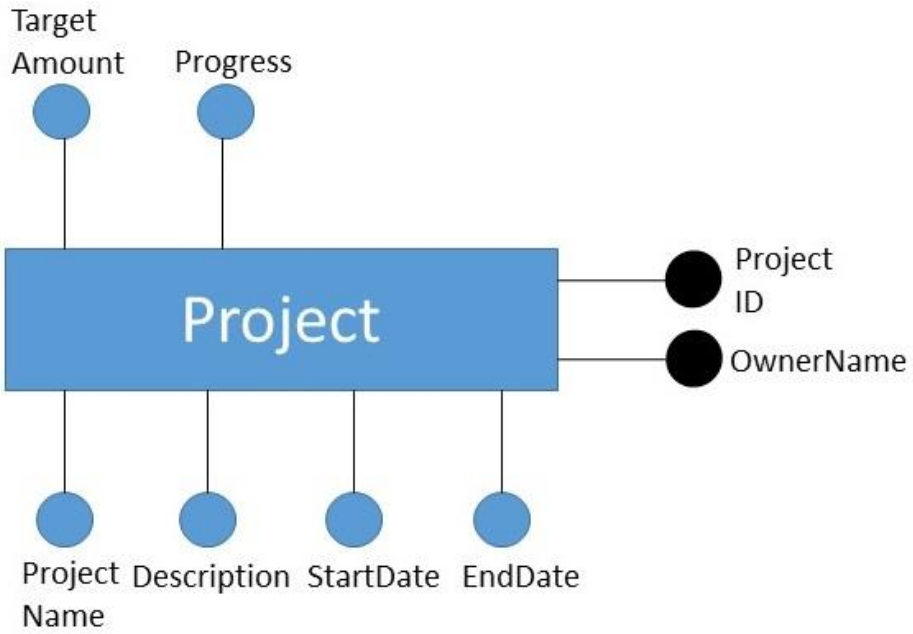
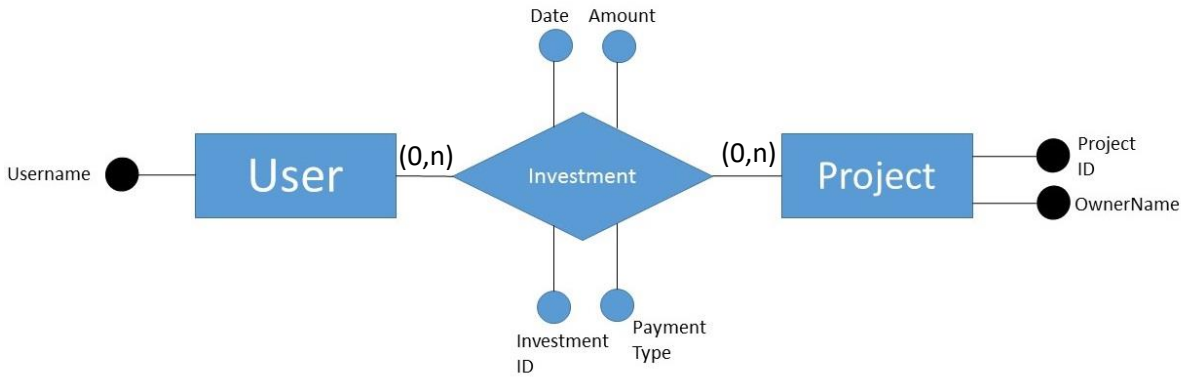


Fig 3. Investment Relationship



Relational Schema

User Entity

```
create table users (
        UID int,
        userName varchar(50),
        pssword varchar(50),
        dateJoined date,
        isAdmin boolean NOT NULL,
        isBanned boolean NOT NULL,
        billingAddress varchar(100),
        PRIMARY KEY (userName)
    );
```

Project Entity

```
create table projectsOwnership (
        projectName varchar(100) NOT NULL,
        projectDescription varchar(1000),
        startDate date NOT NULL,
        endDate date NOT NULL,
        projectID varchar(50),
        ownerName varchar(10)NOT NULL,
        targetAmount int NOT NULL CHECK (targetAmount > 0),
        progress int NOT NULL CHECK (progress >= 0),
        category varchar(20),
        projectStatus varchar(10),
        PRIMARY KEY (projectID, ownerName),
        FOREIGN KEY (ownerName) REFERENCES users(userName)
        on update cascade);
```

Investment Relationship between User and Project

```
create table investments (
        amount int NOT NULL CHECK (amount > 0),
        dateInvested date NOT NULL,
        investmentID varchar(100),
        investorName varchar(50) NOT NULL,
        investmentType varchar(50) NOT NULL,
        projectID varchar(50) NOT NULL,
        ownerName varchar(50) NOT NULL,
        CONSTRAINT chk_InvestmentType CHECK (investmentType IN
        ('eNETS','Paypal', 'Credit Card')),
        PRIMARY KEY (investmentID),
        FOREIGN KEY (projectID, ownerName) REFERENCES
         projectsOwnership(projectID, ownerName)
        on delete cascade
        on update cascade,
        FOREIGN KEY (investorName) REFERENCES users(userName)
        on delete cascade
        on update cascade);
```

Integrity Constraints

1. Data Type constraints:

We need some attributes in the relations to follow certain obvious data type constraints. For example, for crowd funding projects created by users, the target amount to be raised must clearly be more than 0. We use table constraints to achieve this:

```
targetAmount int NOT NULL CHECK (targetAmount > 0)
```

When such constraints are broken by users, an exception will be raised in the database which will be converted to relevant feedback on the website.

Other examples of data type constraints during creation of table:

```
CONSTRAINT chk_InvestmentType CHECK (investmentType IN ('eNETS',
'Paypal', 'Credit Card'))

progress int NOT NULL CHECK (progress >= 0)
```

2. Foreign Key Constraints:

2.1 Project

We identify crowdfunding projects as weak entity – it cannot exist on its own without a user owner. As such, we use define the owner username and project identification number (ID) as the primary key used to identify projects (Note that ID alone will not allow the identification of projects, much alike the situation of repeated matriculation numbers of students across different universities) Under such circumstances, the owner username naturally becomes a foreign key constraint – the owner of a crowdfunding project must be a registered user under the User schema:

```
FOREIGN KEY (ownerName) REFERENCES users(userName)
```

2.2 Investment

Investments, being a relationship, clearly holds the primary keys of User (investors) and Projects. These keys become foreign keys under the Investment Schema:

```
FOREIGN KEY (projectID, ownerName) REFERENCES
projectsOwnership(projectID, ownerName)

FOREIGN KEY (investorName) REFERENCES users(userName)
```

3. Primary Key Constraints for Entity:

3.1 User

Because users are required to log into the website with their username, each username will be unique to a user. As such, it becomes straightforward to use the username as the Primary Key under the User schema (See *Fig 1*). The snippet of code is as follows:

```
PRIMARY KEY (userName)
```

3.2 Project

It would be unwise to use Project description or name as the Primary key (unlike users) because projects can possibly have the same chosen name. As such, we need to impose a project ID for each project as part of the Primary key. However, because we have identified Project as a weak entity which is dependent on User (since a User owns a Project), we additionally include the User's Primary key as part of the Project Primary key. The resulting Primary key is as follows:

```
PRIMARY KEY (projectID, ownerName)
```

Advanced SQL features

- ## Triggers and Functions

We use the *Trigger and Function* feature of SQL to automate updates to the database when actions are performed by users. In some situations, *Triggers* can also be used to enforce certain constraints which cannot be implemented when creating the Relational Schema. For example, when a user attempts to change his password into the same password (this cannot be created as a SQL constraint directly), the following *Trigger* is able to reject this action by raising an exception in the database, which is then translated to feedback to the user:

```
create trigger samePasswordError
  before update
  on users
  for each ROW
  when (old.pssword = new.pssword)
  EXECUTE PROCEDURE samePassword();

create or replace function samePassword()
   returns trigger as $$
   BEGIN
           raise exception 'New password must be different;
           return null;
   END;
   $$ language plpgsql;
```

Please refer to source code for other examples of *Triggers and Functions.*

- ## Aggregation

Because users can invest in various projects and projects can receive investments from different users, it is then natural to use *aggregates* in SQL over such attributes. Various functionalities on the website are enabled with the use of aggregate functions. For example, an admin can check the highest investment on a certain date; the query is as follows:

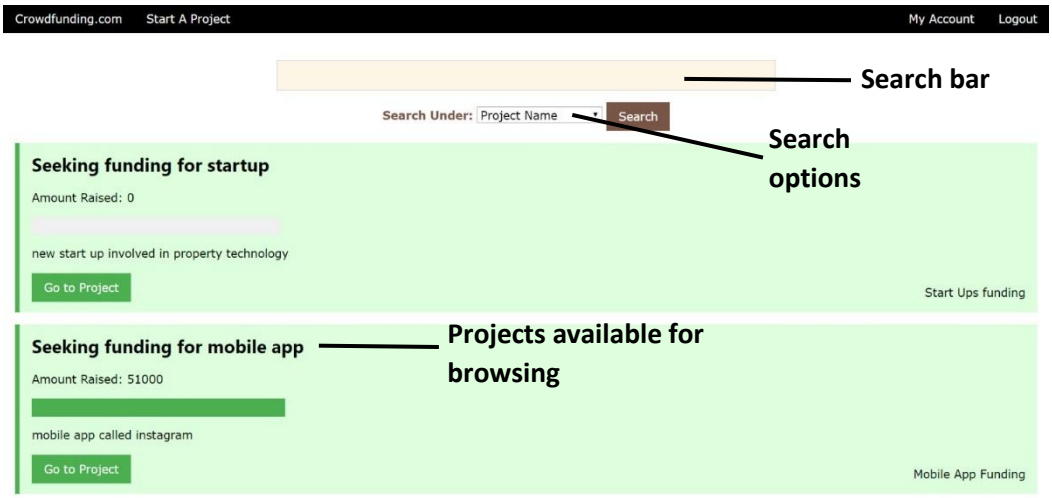
```
SELECT MAX(investments.Amount) AS max FROM investments WHERE Date =
%ChosenDate%
```

A user can also choose to view his total investment with this query:

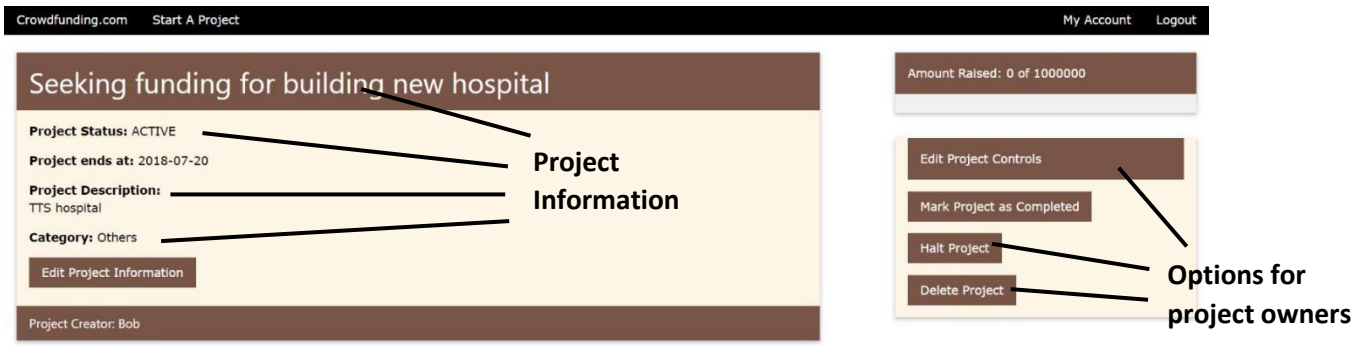
```
SELECT SUM(amount) AS sum FROM investments WHERE investorName = %name
```

Application screenshot

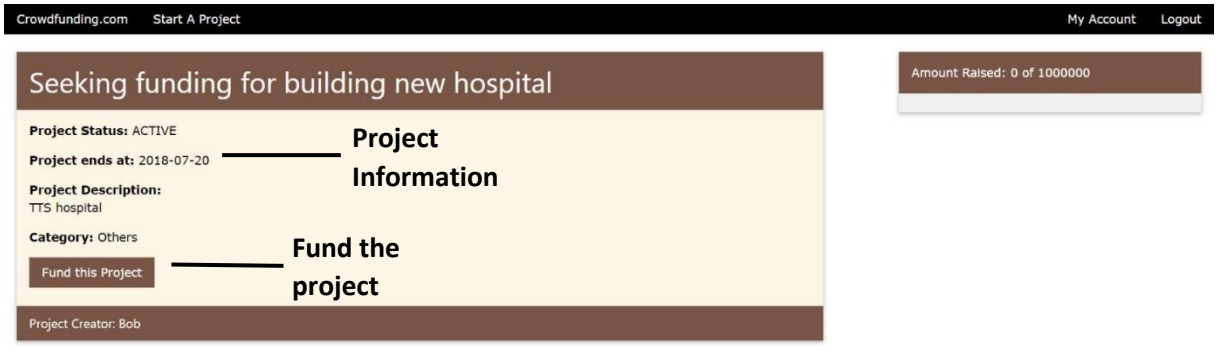
Main Page



Project View Page for Project Owner



Project View Page for other Users



User Profile Page