Bilkent University

CS353 Group 3

Zebra

Project Design Report

Group Members:

Omer Faruk Bababemez	21601216
Idıl Hanhan	21601289
Ensar Kaya	21502089
Berfin Kucuk	21502396

January 30, 2020

Contents

1 Final Entity/Relationship Diagram

1.1 Revisions

The E/R diagram has been revised according to the feedback that is received from Mustafa Çavdar.

- Added a relation between *User* and *Device*.
- Deleted storage attribute from *Device* and added model attribute to *Device*.
- Added name attribute to the *possess* relationship between *User* and *Device*.
- Added user_version attribute to *owns* relationship between *User* and *Device*.
- Added *add_wish* relationship between *User* and *Application*.
- Changed the relation name between *User* and *Review* for more clarity.
- Changed the cardinality between *Editor* and *Restriction*.
- Added total participation to the following relations.
  - *Wallet* side of the *has*.
  - *Card* side of the *includes*.
  - *Restriction* side of the *contains*.
  - *Request* side of the *approves*.
  - *Restriction* side of the *given_by*.
- Primary key added to *Wallet* and *Review*.
- Added *id* as primary key to the relation *Restriction*.
- Removed *age* attribute from the user.
- Reduced number of primary keys of attribute *Account*. *email* is not a primary key, but we specified it as uniqueness in section two, Relational Schema.
- Attribute *name* is added to the *Card* table to indicate the name on the credit cards.

- Removed relationship between *Developer* and *Admin*. This means that the Developers will sign up to Zebra themselves and will not be signed up by an Admin.

1.2 Revised Entity/Relationship Diagram

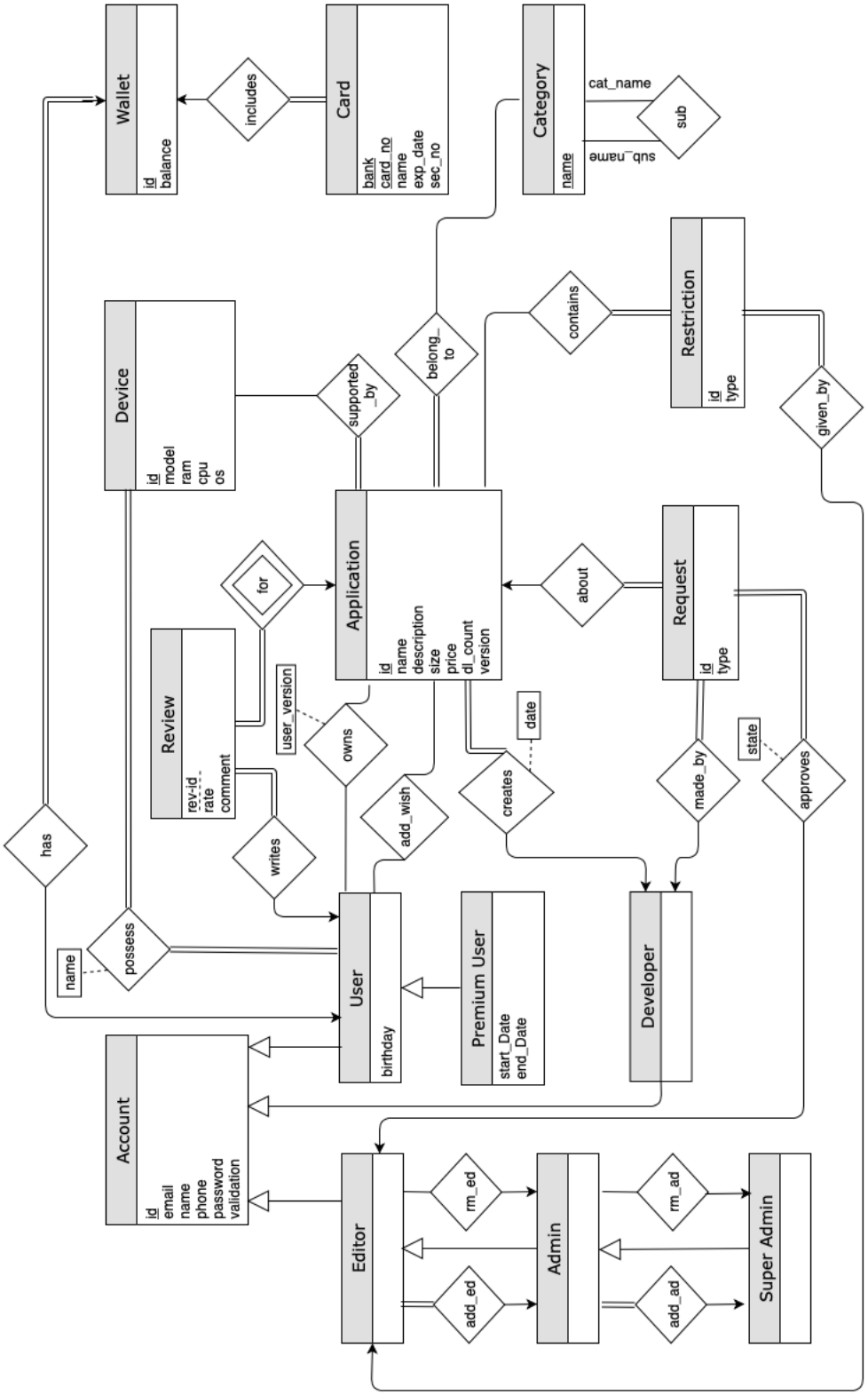

Figure 1: Entity Relationship diagram for Zebra [1][2]

2 Relational Schemas

2.1 Account

Relational Model

Account(<u>id</u>, email, name, phone, password, validation)

Functional Dependencies

id → email, name, phone, password, validation

email → id

Candidate Keys

{(id), (email)}

Normal Form

BCNF

Table Definition

```
CREATE TABLE Account(
id INT PRIMARY KEY AUTO_INCREMENT,
name VARCHAR(30)
email VARCHAR(64) NOT NULL UNIQUE,
phone VARCHAR(15) UNIQUE,
password VARCHAR(32) NOT NULL,
validation varchar (50) DEFAULT ' ' ,
) ENGINE = INNODB;
```

2.2 Super Admin

Relational Model

Super_Admin(<u>id</u>)

Functional Dependencies

None

Candidate Keys

{(id)}

Normal Form

BCNF

Table Definition

```
CREATE TABLE Super_Admin(
id INT PRIMARY KEY,
FOREIGN KEY ( id ) REFERENCES Account ( id )
ON DELETE CASCADE
ON UPDATE RESTRICT
) ENGINE = INNODB;
```

2.3 Admin

Relational Model

Admin(<u>id</u>, s-id)

Functional Dependencies

id $\rightarrow$ s-id

Candidate Keys

{(id)}

Normal Form

BCNF

Table Definition

```
CREATE TABLE Admin(
id INT PRIMARY KEY,
s-id INT
FOREIGN KEY ( id ) REFERENCES Account ( id )
ON DELETE CASCADE
ON UPDATE RESTRICT
FOREIGN KEY ( s-id ) REFERENCES Super_Admin ( id )
ON DELETE CASCADE
ON UPDATE RESTRICT
) ENGINE = INNODB;
```

2.4 rm_ad

Relational Model

rm_ad(<u>adm-id</u>, s-id)

Functional Dependencies

adm-id $\rightarrow$ s-id

Candidate Keys

{(adm-id)}

Normal Form

BCNF

Table Definition

```
CREATE TABLE rm_ad(
adm-id INT PRIMARY KEY,
s-id INT,
FOREIGN KEY (adm-id) REFERENCES Admin(id)
    ON DELETE RESTRICT
    ON UPDATE RESTRICT,
FOREIGN KEY (s-id) REFERENCES Super_Admin(id)
    ON DELETE SET NULL
    ON UPDATE RESTRICT
) ENGINE = INNODB;
```

2.5 Editor

Relational Model

Editor(<u>id</u>, adm-id)

Functional Dependencies

id $\rightarrow$ adm-id

Candidate Keys

{(id)}

Normal Form

BCNF

Table Definition

```
CREATE TABLE Editor(
id INT PRIMARY KEY,
adm-id INT,
FOREIGN KEY (id) REFERENCES Account(id)
    ON DELETE CASCADE
    ON UPDATE RESTRICT,
FOREIGN KEY (adm-id) REFERENCES Admin(id)
    ON DELETE SET NULL
    ON UPDATE RESTRICT
) ENGINE = INNODB;
```

2.6 rm_ed

Relational Model

rm_ed(e-id, adm-id)

Functional Dependencies

e-id → adm-id

Candidate Keys

{(e-id)}

Normal Form

BCNF

Table Definition

```
CREATE TABLE rm_ed(
e-id INT PRIMARY KEY,
adm-id INT,
FOREIGN KEY (e-id) REFERENCES Editor(id)
    ON DELETE RESTRICT
    ON UPDATE RESTRICT,
FOREIGN KEY (adm-id) REFERENCES Admin(id)
    ON DELETE SET NULL
    ON UPDATE RESTRICT
) ENGINE = INNODB;
```

2.7 Developer

Relational Model

Developer(<u>id</u>)

Functional Dependencies

None

Candidate Keys

{(id)}

Normal Form

BCNF

Table Definition

```
CREATE TABLE Developer(
id INT PRIMARY KEY,
adm-id INT NOT NULL,
FOREIGN KEY (id) REFERENCES Account(id)
    ON DELETE CASCADE
    ON UPDATE RESTRICT,
) ENGINE = INNODB;
```

2.8 User

Relational Model

User(<u>id</u>, birthday, w-id)

Functional Dependencies

id → birthday, w-id

w-id → id

Candidate Keys

{(id),(w-id)}

Normal Form

BCNF

Table Definition

```
CREATE TABLE User(
id INT PRIMARY KEY,
birthday CHAR(8),
w-id INT UNIQUE,
FOREIGN KEY ( id ) REFERENCES Account ( id )
ON DELETE CASCADE
ON UPDATE RESTRICT
FOREIGN KEY ( w-id ) REFERENCES Wallet( id )
ON DELETE CASCADE
ON UPDATE RESTRICT
) ENGINE = INNODB;
```

2.9 Premium User

Relational Model

Premium_User(id, start_Date, end_Date)

Functional Dependencies

id → start_Date, end_Date

Candidate Keys

{(id)}

Normal Form

BCNF

Table Definition

```
CREATE TABLE Premium_User(
id INT PRIMARY KEY,
start_Date DATE,
end_Date DATE,
FOREIGN KEY ( id ) REFERENCES User( id )
ON DELETE CASCADE
ON UPDATE RESTRICT
) ENGINE = INNODB;
```

2.10 Wallet

Relational Model

Wallet(<u>id</u>, balance)

Functional Dependencies

id $\rightarrow$ balance

Candidate Keys

{(id)}

Normal Form

BCNF

Table Definition

```
CREATE TABLE Wallet(
id INT PRIMARY KEY,
balance INT DEFAULT 0,
FOREIGN KEY (u-id) REFERENCES User(id)
    ON DELETE CASCADE
    ON UPDATE RESTRICT,
) ENGINE = INNODB;
```

2.11 Card

Relational Model

Card(bank,card-no, name, exp_date, sec_no, w-id)

Functional Dependencies

bank,card-no → exp_date, sec_no, w-id, name

Candidate Keys

{(bank,card-no)}

Normal Form

BCNF

Table Definition

```
CREATE TABLE Card(
bank VARCHAR(20) PRIMARY KEY,
card-no INT PRIMARY KEY,
name VARCHAR(50),
exp_date DATE NOT NULL,
sec_no INT NOT NULL,
w-id INT NOT NULL,
FOREIGN KEY (w-id) REFERENCES Wallet(id)
    ON DELETE CASCADE
    ON UPDATE RESTRICT,
) ENGINE = INNODB;
```

2.12 Application

Relational Model

Application(id, name, description, size, price, dl_count, version, dev-id, date)

Functional Dependencies

id → name, description, size, price, dl_count, version, dev-id, date

Candidate Keys

{(id)}

Normal Form

BCNF

Table Definition

```
CREATE TABLE Application(
id INT PRIMARY KEY,
name VARCHAR(20),
description VARCHAR(150),
size FLOAT (15,4),
price FLOAT (10,2),
dl_count INT,
version FLOAT(15,4),
dev-id INT NOT NULL,
date DATE,
FOREIGN KEY (dev-id) REFERENCES Developer(id)
    ON DELETE CASCADE
    ON UPDATE RESTRICT,
) ENGINE = INNODB;
```

2.13 owns

Relational Model

owns(u-id, app-id,user-version)

Functional Dependencies

u-id, app-id→ user-version

Candidate Keys

{(u-id,app-id)}

Normal Form

BCNF

Table Definition

```
CREATE TABLE owns(
u-id INT PRIMARY KEY,
app-id INT PRIMARY KEY,
user-version FLOAT(10,2),
FOREIGN KEY (u-id) REFERENCES User(id)
    ON DELETE CASCADE
    ON UPDATE RESTRICT,
FOREIGN KEY (app-id) REFERENCES Application(id)
    ON DELETE CASCADE
    ON UPDATE RESTRICT,
) ENGINE = INNODB;
```

2.14 privileged_access

Relational Model

privileged_access(p_id, app_id, discount_rate)

Functional Dependencies

p_id, app_id→ discount_rate

Candidate Keys

{(p_id, app_id)}

Normal Form

BCNF

Table Definition

```
CREATE TABLE privileged_access(
p_id INT,
app_id INT,
discount_rate INT,
FOREIGN KEY (p-id) REFERENCES Premium_User(id)
    ON DELETE CASCADE
    ON UPDATE RESTRICT,
FOREIGN KEY (app-id) REFERENCES Application(id)
    ON DELETE CASCADE
    ON UPDATE RESTRICT,
PRIMARY KEY (p_id, app_id)
) ENGINE = INNODB;
```

2.15 add_wish

Relational Model

add_wish(u-id, app-id)

Functional Dependencies

None

Candidate Keys

{(u-id,app-id)}

Normal Form

BCNF

Table Definition

```
CREATE TABLE add_wish(
u-id INT PRIMARY KEY,
app-id INT PRIMARY KEY,
FOREIGN KEY (u-id) REFERENCES User(id)
    ON DELETE CASCADE
    ON UPDATE RESTRICT,
FOREIGN KEY (app-id) REFERENCES Application(id)
    ON DELETE CASCADE
    ON UPDATE RESTRICT,
) ENGINE = INNODB;
```

2.16 Review

Relational Model

Review(rev-id,app-id, rate, comment, u-id)

Functional Dependencies

rev_id, app_id → rate, comment, u-id

Candidate Keys

{(rev-id,app-id)}

Normal Form

BCNF

Table Definition

```
CREATE TABLE Review(
rev-id INT PRIMARY KEY,
app-id INT PRIMARY KEY,
rate INT,
comment VARCHAR(150),
u-id INT NOT NULL,
FOREIGN KEY (u-id) REFERENCES User(id)
    ON DELETE CASCADE
    ON UPDATE RESTRICT,
FOREIGN KEY (app-id) REFERENCES Application(id)
    ON DELETE CASCADE
    ON UPDATE RESTRICT,
) ENGINE = INNODB;
```

2.17 Device

Relational Model

Device(id, ram, cpu, os, model)

Functional Dependencies

id → ram, cpu, os, model

Candidate Keys

{(id)}

Normal Form

BCNF

Table Definition

```
CREATE TABLE Device(
id INT PRIMARY KEY,
model VARCHAR(30) UNIQUE,
ram INT,
cpu VARCHAR(30),
os VARCHAR(30),
STORAGE sizeFLOAT(15,4)
) ENGINE = INNODB;
```

2.18 possess

Relational Model

possess(u-id, dvc-id, name)

Functional Dependencies

u-id, dvc-id → name

Candidate Keys

{(u-id,dvc-id)}

Normal Form

BCNF

Table Definition

```
CREATE TABLE possess(
u-id INT PRIMARY KEY,
dvc-id INT PRIMARY KEY,
name VARCHAR(50),
FOREIGN KEY (u-id) REFERENCES User(id)
    ON DELETE CASCADE
    ON UPDATE RESTRICT,
FOREIGN KEY (dvc-id) REFERENCES Device(id)
    ON DELETE CASCADE
    ON UPDATE RESTRICT,
) ENGINE = INNODB;
```

2.19 supported_by

Relational Model

possess(app-id, dvc-id)

Functional Dependencies

None

Candidate Keys

{(app-id,dvc-id)}

Normal Form

BCNF

Table Definition

```
CREATE TABLE supported_by(
app-id INT PRIMARY KEY,
dvc-id INT PRIMARY KEY,
FOREIGN KEY (app-id) REFERENCES Application(id)
    ON DELETE CASCADE
    ON UPDATE RESTRICT,
FOREIGN KEY (dvc-id) REFERENCES Device(id)
    ON DELETE CASCADE
    ON UPDATE RESTRICT,
) ENGINE = INNODB;
```

2.20 Request

Relational Model

Request(<u>id</u>, type, app-id, dev-id, e-id, state)

Functional Dependencies

id → type, app-id, dev-id, e-id, state

Candidate Keys

{(id)}

Normal Form

BCNF

Table Definition

```
CREATE TABLE Request(
id INT PRIMARY KEY,
dev-id INT NOT NULL,
app-id INT NOT NULL,
e-id INT NOT NULL,
state varchar(30),
FOREIGN KEY (dev-id) REFERENCES Developer(id)
    ON DELETE CASCADE
    ON UPDATE RESTRICT,
FOREIGN KEY (app-id) REFERENCES Application(id)
    ON DELETE CASCADE
    ON UPDATE RESTRICT,
FOREIGN KEY (e-id) REFERENCES Editor(id)
    ON DELETE SET NULL
    ON UPDATE RESTRICT,
) ENGINE = INNODB;
```

2.21 Restriction

Relational Model

Restriction(<u>id</u>, type, e-id)

Functional Dependencies

id $\rightarrow$ type, e-id

Candidate Keys

{(id)}

Normal Form

BCNF

Table Definition

```
CREATE TABLE Restriction(
id INT PRIMARY KEY,
type VARCHAR(30),
e-id INT,
FOREIGN KEY (e-id) REFERENCES Editor(id)
    ON DELETE SET NULL
    ON UPDATE RESTRICT,
) ENGINE = INNODB;
```

2.22 contains

Relational Model

contains(r-id, app-id)

Functional Dependencies

None

Candidate Keys

{(r-id, app-id)}

Normal Form

BCNF

Table Definition

```
CREATE TABLE contains(
r-id INT PRIMARY KEY,
app-id INT PRIMARY KEY,
FOREIGN KEY (r-id) REFERENCES Restriction(id)
    ON DELETE CASCADE
    ON UPDATE RESTRICT,
FOREIGN KEY (app-id) REFERENCES Application(id)
    ON DELETE CASCADE
    ON UPDATE RESTRICT,
) ENGINE = INNODB;
```

2.23 Category

Relational Model

Category(<u>name</u>)

Functional Dependencies

None

Candidate Keys

{(name)}

Normal Form

BCNF

Table Definition

```
CREATE TABLE Category(
name VARCHAR(30) PRIMARY KEY,
) ENGINE = INNODB;
```

2.24 sub

Relational Model

sub(sub_name,cat_name)

Functional Dependencies

None

Candidate Keys

{(sub_name,cat_name)}

Normal Form

BCNF

Table Definition

```
CREATE TABLE sub(
sub_name VARCHAR(30) PRIMARY KEY,
cat_name VARCHAR(30) PRIMARY KEY,
FOREIGN KEY (cat_name) REFERENCES Category(name)
    ON DELETE CASCADE
    ON UPDATE RESTRICT,
FOREIGN KEY (sub_name) REFERENCES Category(name)
    ON DELETE CASCADE
    ON UPDATE RESTRICT,
) ENGINE = INNODB;
```

2.25 belong_to

Relational Model

belong_to(c_name, app-id)

Functional Dependencies

None

Candidate Keys

{(c-name, app-id)}

Normal Form

BCNF

Table Definition

```
CREATE TABLE contains(
c-name INT PRIMARY KEY,
app-id INT PRIMARY KEY,
FOREIGN KEY (c_name) REFERENCES Category(name)
    ON DELETE CASCADE
    ON UPDATE RESTRICT,
FOREIGN KEY (app-id) REFERENCES Application(id)
    ON DELETE CASCADE
    ON UPDATE RESTRICT,
) ENGINE = INNODB;
```

3 Normalization of Tables

The tables are all in at least 3NF, thus no normalization is required.

4 Functional Components

4.1 Use Case Diagram

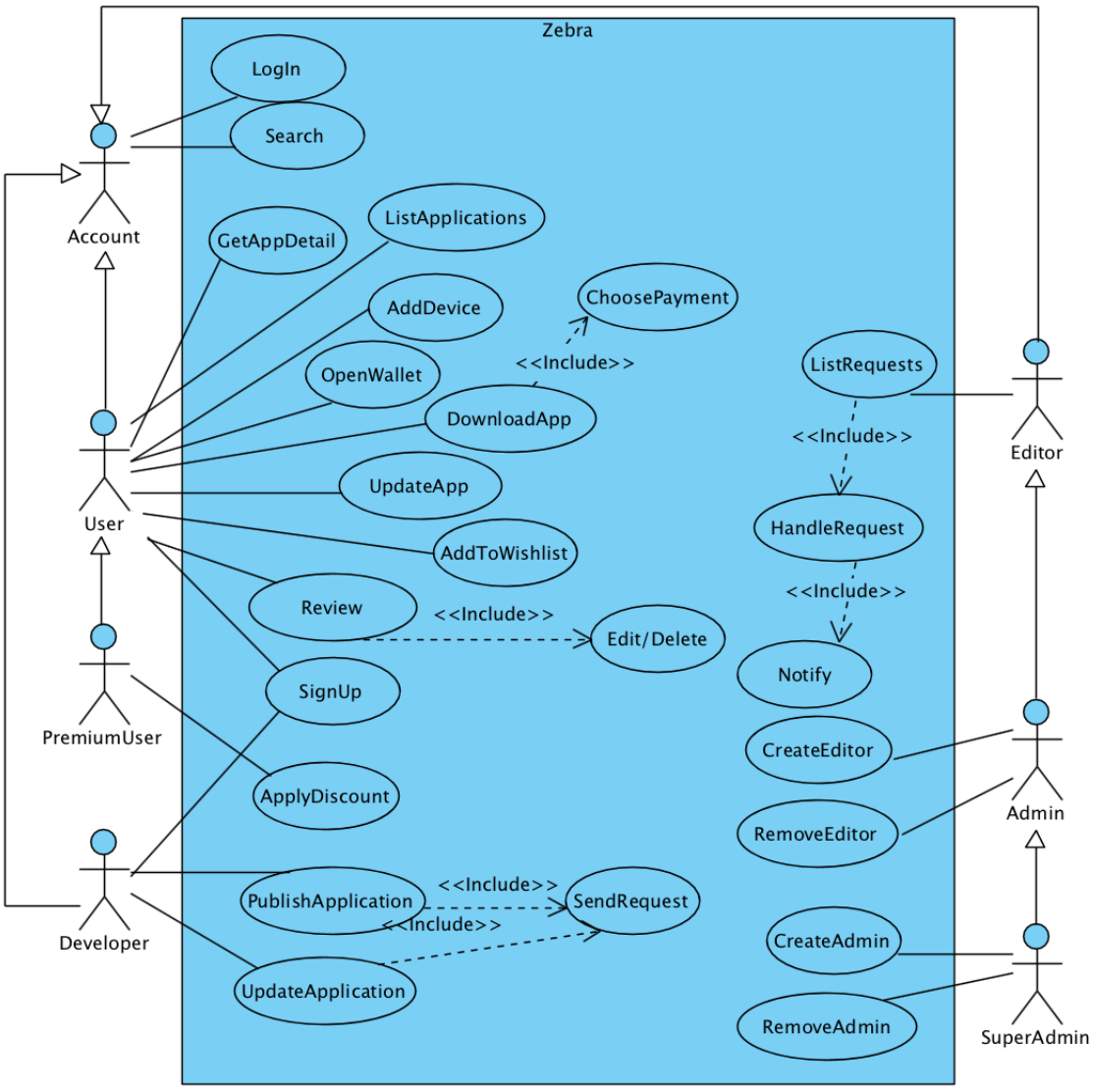

Figure 2: Use Case Diagram for Zebra [1]

4.2 Scenarios

Log In

Use Case Name:	Login
Participating Actor:	Account
Entry Condition:	• Account owner entered the website • Account owner has a verified account
Exit Condition:	• Account owner is logged in.
Flow of Events:	
1.Basic Flow:	
1.1. Account owner entered the website and clicked on Login.	
	1.2. The system displays the login form.
1.1. Account owner enters their valid email and password.	
	1.2. The system checks the given email and password. When they are found to be correct, the system displays the home page to the Account owner.
2. Alternative Flows:	
2.1. Account owner enters invalid email and password	
	2.2. The given username and password are incorrect, the system prompts the account owner to fill their information again.
2.3. Account owner enters their valid email and password.	
	2.4. The system checks the given email and password. When they are found to be correct, the system displays the home page.

Figure 3: Scenario for log in

Search

Use Case Name:	Search
Participating Actor:	Account
Entry Condition:	• Account is logged in to Zebra • Account selected the search bar.
Exit Condition:	• Account owner sees the result of their search
Flow of Events:	
1.Basic Flow:	
1.1. Account owner writes a keyword they want to search to the search bar.	
	1.2. The system finds the applications related to the input the account has given. Then displays these applications on the page.

Figure 4: Scenario for search

Sign Up

Use Case Name:	SignUp
Participating Actor:	User / Developer
Entry Condition:	• User/Developer entered the website • User clicked on SignUp
Exit Condition:	• User / Developer signed up to Zebra
Flow of Events:	
1.Basic Flow:	
1.1. User/Developer entered the website and clicked on SignUp	
	1.2. The system displays the sign up form.
1.3. User/Developer fills the sign up form with their name, email, phone number and password and whether they are a Developer or User.	
	1.4. The system gets the relevant information and creates a new user / developer account with this information. The system prompts the user/developer to go to the login page.
2. Alternative Flows	
2.1. User/Developer fills the sign up form with their information.	
	2.2. The system gets the relevant information and finds that a User/ Developer account with this information already exists. The system prompts the user about the failure of the sign up.
2.3. The User fills the sign up form with valid information.	
	2.4. The system gets the relevant information and creates a new user / developer account with this information. The system prompts the user/developer to go to the login page.

Figure 5: Scenario for sign up

List Applications

Use Case Name:	ListApplications
Participating Actor:	User
Entry Condition:	• User is logged in to Zebra • User selected a category or another list of applications or purchased applications.
Exit Condition:	• User sees a list of applications
Flow of Events:	
1.Basic Flow:	
1.1. The User selects a category or a list of applications such as new arrivals or top applications or purchased applications.	
	1.2. The system gets every application relevant to the User's choice and displays them on the page.

Figure 6: Scenario for list applications

Get App Detail

Use Case Name:	GetAppDetail
Participating Actor:	User
Entry Condition:	• User is logged in to Zebra • User has selected a single application
Exit Condition:	• User sees the applications page
Flow of Events:	
1.Basic Flow:	
1.1. User selects a single application from a list of applications or search result.	
	1.2. The system gets the information related to that application (such as app description, creator, requirements, download count, average rating and reviews) and displays them as a separate application page.

Figure 7: Scenario for get app detail

Download App

Use Case Name:	DownloadApp
Participating Actor:	User
Entry Condition:	• User is logged in to Zebra • User clicked on the download
Exit Condition:	• User downloads the application
Flow of Events:	
1.Basic Flow:	
1.1. User clicks on the download button next to each application on a list of applications or on the single page of the application.	
	1.2. The system gets the condition that must be satisfied, such as device type, in order to download the application. Then the system prompts the user on which devices they can download the said application to.
1.3. User has a compatible device so they continue with the download.	
	1.4. If the application requires payment, the system displays the checkout page.
1.5. User pays the application price with the money they loaded to their wallet or with a card they saved in their wallet or they enter new card information.	
	1.6. The system downloads the application to the User's device.
2.Alternative Flow:	
2.1. User clicks on the download button	
	2.2. The system gets the condition that must be satisfied, such as device type, in order to download the application. Then the system prompts the user on which devices they can download the said application to.
2.3. User does not have a compatible device so the download is cancelled.	
Special/Quality Requirements:	• DownloadApp use case **includes** ChoosePayment use case.

Figure 8: Scenario for downloading an application

Open Wallet

Use Case Name:	OpenWallet
Participating Actor:	User
Entry Condition:	• User is logged in to Zebra • User clicked on their wallet
Exit Condition:	• New card added to wallet or the balance of the wallet is updated
Flow of Events:	
1.Basic Flow:	
1.1. User clicks on their wallet.	
	1.2. The systems displays the Wallet page.
1.3.1 If User selects Add Credit Card	
	1.3.1.2 The system displays the form for adding a credit card.
1.3.1.3 User fills in the form with their credit card information and clicks submit	
	1.3.1.4 The system adds the new credit card to the wallet
1.3.2 If the User selects Load Money	
	1.3.2.2 The system displays the credit cards saved in the wallet and a form to specify the amount to load.
1.3.2.3 User selects the card they want to load money from, enters the amount and clicks load.	
	1.3.2.3 The system updates the balance according to the amount specified by the User.

Figure 9: Scenario for opening the wallet

Update App

Use Case Name:	UpdateApp
Participating Actor:	User
Entry Condition:	• User is logged in to Zebra • User has selected the Update page
Exit Condition:	• User updated their application
Flow of Events:	
1.Basic Flow:	
1.1. User selects the Update page.	
	1.2. The system displays the current updates for all of the applications User has downloaded to their device.
1.3. User chooses the application they want to update.	
	1.4. The system updates the application chosen by the User.

Figure 10: Scenario for updating an application

Add Device

Use Case Name:	AddDevice
Participating Actor:	User
Entry Condition:	• User is logged in to Zebra • User is on the Devices page
Exit Condition:	• User added a new device
Flow of Events:	
1.Basic Flow:	
1.1. User goes to the Devices page.	
	1.2. The system displays the devices owned by the User and a form that can be used to add a new device.
1.3. User fills the form with the information of the device they want to add. This information includes device name, RAM, CPU and OS.	
	1.2. The system gets the information and adds the new device to the system. The updated Device page is shown to the user.

Figure 11: Scenario for adding a device

Add to Wishlist

Use Case Name:	AddToWishlist
Participating Actor:	User
Entry Condition:	• User is logged in to Zebra • User has clicked on the AddToWishlist button
Exit Condition:	• The User added the application to their wishlist.
Flow of Events:	
1.Basic Flow:	
1.1. The User clicks on the AddToWishlist button located in the app page for each application.	
	1.2. The system adds the application to the User's wishlist.
2.Alternative Flow:	
2.1. The User clicks on the AddToWishlist button on the applications page.	
	2.2. The system finds that the application is already added to the wishlist and does not make any change.

Figure 12: Scenario for adding an application to wishlist

Review

Use Case Name:	Review
Participating Actor:	User
Entry Condition:	• User is logged in to Zebra • User downloaded the application they want to review
Exit Condition:	• Review of the User is published
Flow of Events:	
1.Basic Flow:	
1.1. The User goes to the review section located in the application's page. The user writes their review and/or rates the application.	
	1.2. The system updates the application page to include the review of the User.
2.Alternative Flow:	
2.1. The User clicks on their past review	
	2.2.The system shows their review in editable format
2.3. The user edits or deletes their review.	
Special/Quality Requirements:	• Review use case **includes** Edit/Delete use case.

Figure 13: Scenario for reviewing an application

Apply Discount

Use Case Name:	ApplyDiscount
Participating Actor:	PremiumUser
Entry Condition:	• PremiumUser is logged in to Zebra • PremiumUser selected a paid application to download
Exit Condition:	• PremiumUser applied a discount to the price of a paid application
Flow of Events:	
1.Basic Flow:	
1.1. PremiumUser selects a paid application to download. During the downloading process the PremiumUser clicks on the discount option.	
	1.2. The system checks if the PremiumUser is qualified to get the discount. If yes, the system applies the discount and shows the new price to the user.
1.3. PremiumUser pays the discounted price.	
	1.4. The system downloads the application to PremiumUser's device.

Figure 14: Scenario applying discount

Publish Application

Use Case Name:	PublishApplication
Participating Actor:	Developer
Entry Condition:	• Developer is logged in to Zebra • Developer clicked on PublishApplication
Exit Condition:	• Developer made a request to publish a new application
Flow of Events:	
1.Basic Flow:	
1.1. Developer clicks on the Publish Application button.	
	1.2. The system shows the form for publishing applications.
1.3. The Developer fills the form with application information such as name and description and submits the form.	
	1.4. The system accepts the new application publish as a request that requires approval.
Special/Quality Requirements:	• PublishApplication use case **includes** SendRequest use case.

Figure 15: Scenario for publishing a new application

Update Application

Use Case Name:	UpdateApplication
Participating Actor:	Developer
Entry Condition:	• Developer is logged in to Zebra • Developer clicked on UpdateApplication
Exit Condition:	• The Developer made a request to update an existing application
Flow of Events:	
1.Basic Flow:	
1.1. The Developer clicks on the Update Application button.	
	1.2. The system shows the Developer a list of their previous applications.
1.3. The Developer selects the application for which they want to issue an update.	
	1.4. The system asks for further information about the update
1.5. The Developer fills the form for updating applications.	
	1.6. The system accepts the update application as a request that requires approval.
Special/Quality Requirements:	• UpdateApplication use case **includes** SendRequest use case.

Figure 16: Scenario for updating an existing application

List Requests & Handle Request

Use Case Name:	ListRequests
Participating Actor:	Editor
Entry Condition:	• Editor is logged in to Zebra
Exit Condition:	• Editor approves or rejects the request
Flow of Events:	
1.Basic Flow:	
1.1. Editor goes to their home page.	
	1.2. The system displays all of the requests that are currently waiting for evaluation.
1.3. Editor selects one of the requests	
	1.4. The system displays the details of the request, namely who made the request and what the request is about
1.5. The request is related to publishing or updating an application, Editor goes over the details of the application and either approves or rejects the request made by a Developer.	
	1.6. The system records the request as handled and allows the application to be published or updated. Then sends a notification to the Developer who made the request.
Special/Quality Requirements:	• ListRequests use case **includes** HandleRequest use case. • HandleRequest use case **includes Notify** use case.

Figure 17: Scenario for listing requests and handling a request

Create Editor

Use Case Name:	CreateEditor
Participating Actor:	Admin
Entry Condition:	• Admin is logged in to Zebra • Admin selected the CreateEditor option
Exit Condition:	• Admin created an Editor
Flow of Events:	
1.Basic Flow:	
1.1. Admin selects the CreateEditor.	
	1.2. The system displays the form for creating a new editor.
1.3. Admin fills the form with the valid information of the Editor they are creating the account for and submits the form.	
	1.4. The system checks if given information is unique. If yes, the system creates and account with this information.
2.Alternative Flow:	
2.1. Admin sends invalid information for the creation of a new account. The information may be invalid because an Editor with the same information exists.	
	2.2. The system checks the validity of the information and finds that it is invalid. The system asks the Admin to give valid information.
2.3. Admin gives a valid information for the creation of a new Editor account.	
	2.4. The system creates an Editor account with given information.

Figure 18: Scenario for creating an editor

Remove Editor

Use Case Name:	RemoveEditor
Participating Actor:	Admin
Entry Condition:	• Admin is logged in to Zebra • Admin selected the RemoveEditor option
Exit Condition:	• Admin removed an Editor
Flow of Events:	
1.Basic Flow:	
1.1. Admin selects the RemoveEditor	
	1.2. The system prompts the Admin to give the information of the Editor to remove.
1.3. Admin sends the email of the Editor that they want to remove.	
	1.4. The system checks if an Editor with given information exists. If yes, that Editor is removed.
2.Alternative Flow:	
2.1. Admin sends an invalid email address to the system. The email may be invalid because it does not exist or because it is the email of another type of account.	
	2.2. The system checks the validity of the email and finds that it is invalid. The system asks the Admin to give a valid email address.
2.3. Admin gives a valid email address for EditorRemoval	
	2.4. The system removes the Editor account with the given email.

Figure 19: Scenario for removing an editor

Create Admin

Use Case Name:	CreateAdmin
Participating Actor:	SuperAdmin
Entry Condition:	• SuperAdmin is logged in to Zebra • SuperAdmin selected the CreateAdmin option
Exit Condition:	• SuperAdmin created an Admin
Flow of Events:	
1.Basic Flow:	
1.1. SuperAdmin selects the CreateAdmin.	
	1.2. The system displays the form for creating a new Admin.
1.3. SuperAdmin fills the form with the valid information of the Admin they are creating the account for and submits the form.	
	1.4. The system checks if given information is unique. If yes, the system creates and account with this information.
2.Alternative Flow:	
2.1. SuperAdmin sends invalid information for the creation of a new account. The information may be invalid because an Admin with the same information exists.	
	2.2. The system checks the validity of the information and finds that it is invalid. The system asks the SuperAdmin to give valid information.
2.3. SuperAdmin gives a valid information for the creation of a new Admin account.	
	2.4. The system creates an Editor account with given information.

Figure 20: Scenario for creating an admin

Remove Admin

Use Case Name:	RemoveAdmin
Participating Actor:	SuperAdmin
Entry Condition:	• SuperAdmin is logged in to Zebra • SuperAdmin selected the RemoveAdmin option
Exit Condition:	• SuperAdmin deleted an Admin
Flow of Events:	
1.Basic Flow:	
1.1. SuperAdmin selects the RemoveAdmin.	
	1.2. The system prompts the SuperAdmin to give the information of the Admin to remove.
1.3. SuperAdmin sends the email of the Admin that they want to remove.	
	1.4. The system checks if an Admin with given information exists. If yes, that Admin is removed.
2.Alternative Flow:	
2.1. SuperAdmin sends an invalid email address to the system for Admin Removal. The email may be invalid because it does not exist or because it is the email of another type of account	
	2.2. The system checks the validity of the email and finds that it is invalid. The system asks the SuperAdmin to give a valid email address.
2.3. SuperAdmin gives a valid email address for AdminRemoval	
	2.4. The system removes the Admin account with the given email.

Figure 21: Scenario for removing an admin

5 User Interface Design and Corresponding SQL Statements

Log in

The page below is the UI design for the log in page for our system.

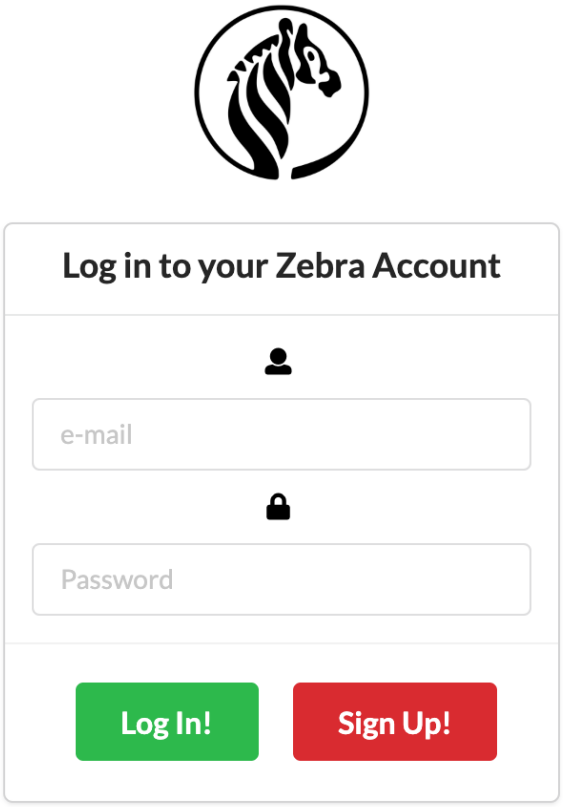

Figure 22: UI design of log in

The corresponding SQL statements are as follows:

Login Validation: We have to get the decrypted password for entered email address. We need ID of the user to verify the type of user for further possible actions in the system. We'll hold ID of the user as @user_id variable. We also need validation status of the account.

```
SELECT id, DECODE(password,'secret'), validation
FROM Account
WHERE email = @email
```

If the validation status is 0 then the system will give a warning. If the validation status is 1 then we will check the type of user since the users of system will be limited in sessions according to their user type. The type checking queries are the followings:

User:

```
SELECT id
FROM User
WHERE id = @user_id
```

Premium User:

```
SELECT id
FROM Premium_User
WHERE id = @user_id
```

Admin:

```
SELECT id
FROM Admin
WHERE id = @user_id
```

Super Admin:

```
SELECT id
FROM Super_Admin
WHERE id = @user_id
```

Editor:

```
SELECT id
FROM Editor
WHERE id = @user_id
```

Developer:

```
SELECT id
FROM Developer
WHERE id = @user_id
```

Sign up

The page below is the UI design for the sign up page for our system.

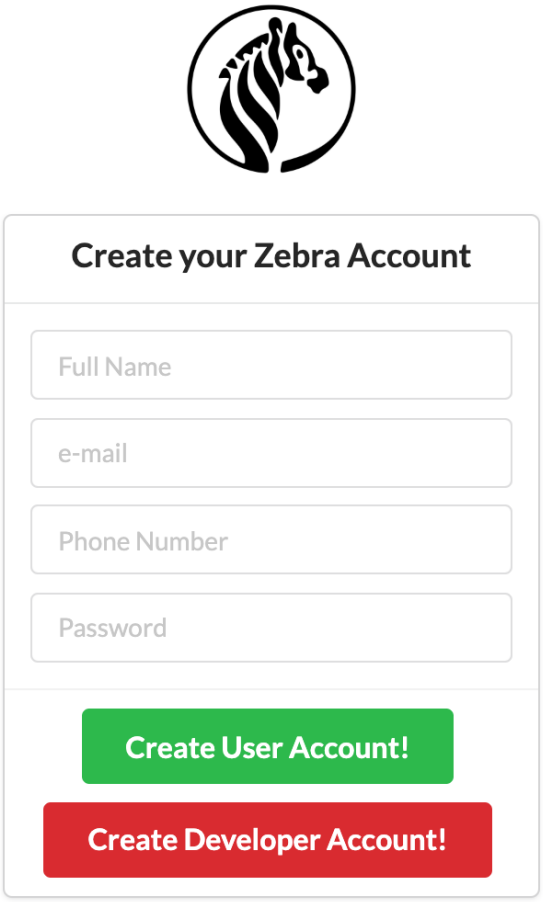

Figure 23: UI design of sign up

The corresponding SQL statements are as follows:

Unique Email Check: We have to check whether the sing up email address already exists in our database or not.

```
SELECT email
FROM Account
WHERE email = @email
```

If the query above returns a non empty set, we will display a warning message and disapprove the sign up attempt. If the query returns an empty set, then it means that the email is a valid email.

At first, we will insert the user into Account table because every signed up user need an ID, with "validation = 0" statement. We will hash the password before adding them into the database and encrypted password will be stored as @password.The next step is generating a verification code and send it to the signed up email address. When the user enters the code correctly in 60 seconds, we will edit the validation status as 1. If the user cannot enter the code in 60 seconds, s/he may try again.

Sign Up with Valid Email:

```
INSERT INTO Account(email, name, phone, password, validation)
VALUES (@email, @name, @phone, ENCODE(@password, 'secret'), 0)
```

Update After the Email Verification:

```
UPDATE Account
SET validation = '1'
WHERE email = @email
```

After adding the new account into Account table and verifying email address, we also need to add it into the User or Developer tables according to their user types. We will store the new account's ID number as @user_id if its a user and @dev_id if its a developer.

```
SELECT id
FROM Account
WHERE email = @email
```

Adding new Account into User Table:

```
INSERT INTO User(id)
VALUES (@user_id)
```

Adding new Account into Developer Table:

```
INSERT INTO Developer(id)
VALUES (@dev_id)
```

This operation to add a developer can be done only by admins and super admins.

User Home Page

The page below is the UI design for the user's home page for our system.

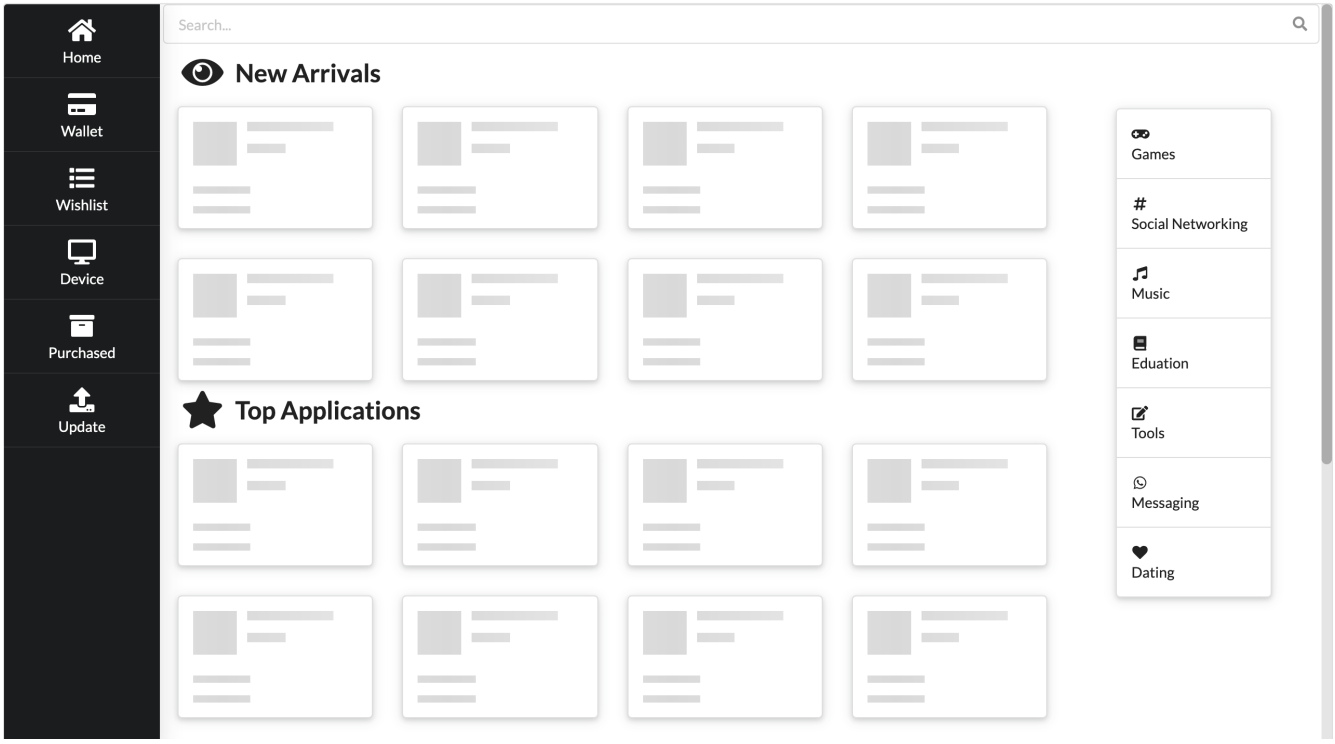

Figure 24: UI design of user's home page

Displaying New Arrivals:

```
SELECT *
FROM Application
WHERE (DATEDIFF(hour,date,GETDATE())/24.0) < 120 AND id IN( SELECT app-id
↪  FROM Request WHERE state = 'approved')
```

The query above will return a table with all applications whose release date is at most 5 days ago.

Displaying Top Applications :

```
SELECT *
FROM Application
WHERE id IN( SELECT app-id FROM Request WHERE state = 'approved')
ORDER BY dl_count DESC limit 25
```

The query above returns all times most downloaded 10 applications.

Filtering According to Categories:

```
SELECT app-id
FROM belong_to
WHERE c-name = @category_name AND app-id IN( SELECT app-id FROM Request WHERE
↪  state = 'approved')
```

The query above returns applications which are classified according to categories. We'll take @category_name as name of the categories such as Games, Music, Education etc

Search Result Page

The page below is the UI design for the user's search result page for our system.

Figure 25: UI design of search result page

When user searches a keyword, the given keyword will be stored in @search_key and the key will be searched in the name and description of all of the applications.

Retrieve the Applications that is Searched with a Search Key:

```
SELECT id
FROM Application
WHERE name like '% @search_key%' OR description like '%@search_key%'
AND id IN( SELECT app-id FROM Request WHERE state = 'approved')
```

Application Page

The page below is the UI design for the application page of our system.

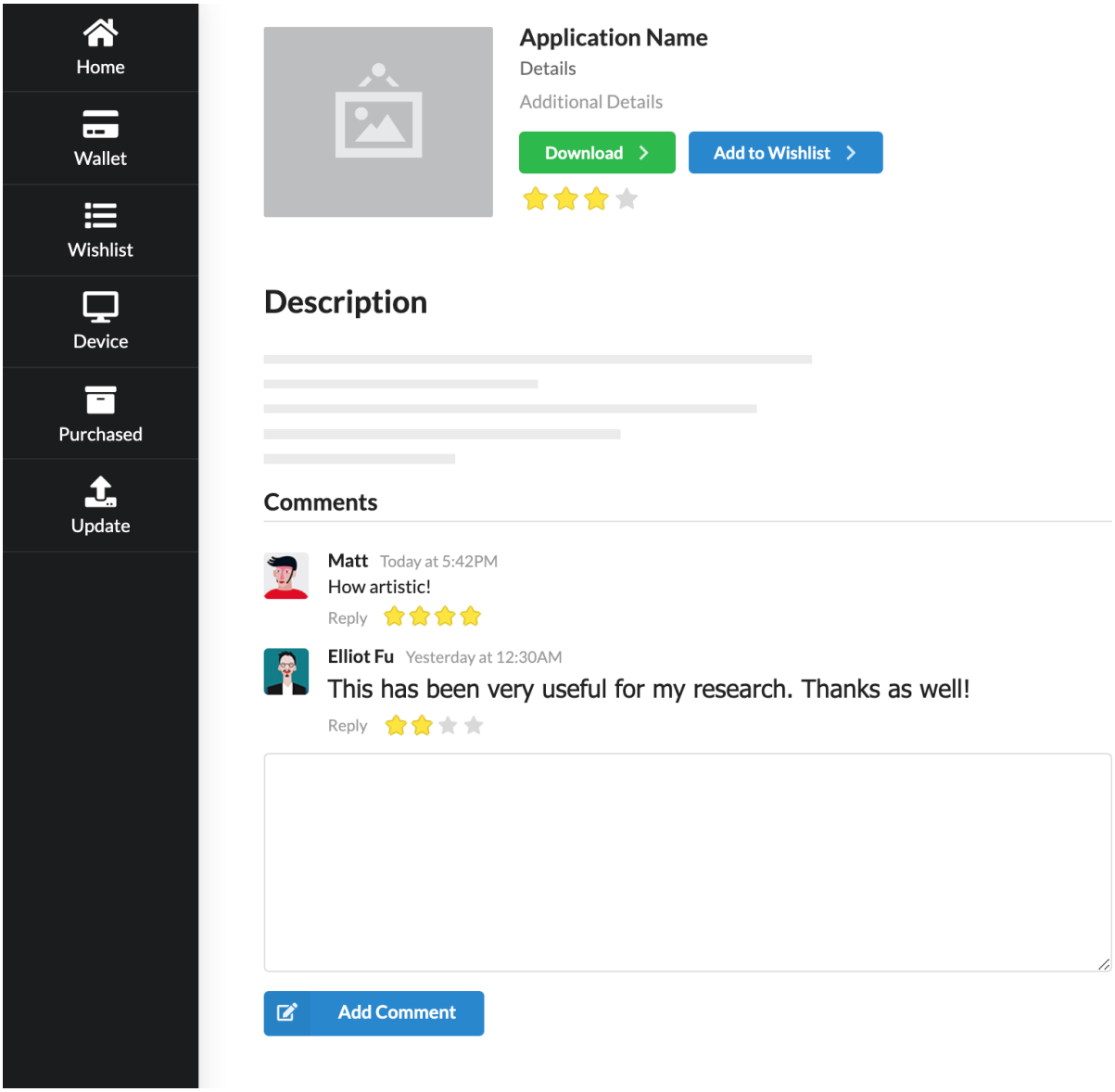

Figure 26: UI design of application page

In this page, we display an application with its description, rate and reviews. The users will be able to leave their reviews of the application in this page. Users can download the application or add the application to their wishlist via this page. If the application is already downloaded, there will be an *Update* button if there is a new version. When the developer of the application visits this page, s/he will see an *Upload New Version* button instead of the *Download* button.

Add an App into Wishlist:

```
INSERT INTO add_wish(u-id, app-id)
VALUES(@user_id, @app_id)
```

Add Review:

```
INSERT INTO Review(rev-id, app-id, rate, comment, u-id)
SELECT @rev-id, app-id, @rate, @comment, @u-id
FROM owns
WHERE u-id = @u-id
```

List Review:

```
SELECT(*)
FROM Review as R
WHERE R.app-id = @app-id
```

Reviews include rates and application rates will be calculated based on these rates.

Application Rates:

```
SELECT AVG(rate)
FROM Review as R
WHERE = R.app-id = @app-id
```

Update App:

```
UPDATE owns O
SET O.user_version = @app-id
```

User can download free or paid apps. If the app is paid, when the user presses the Download button, a pop-up will occur for the payment. The UI for that specific can be seen below.

Download Free App:

```
UPDATE Application
SET dl_count = dl_count + 1
WHERE exists(SELECT app-id, dvc-id
             FROM supported_by)

INSERT INTO owns(u-id, app-id)
VALUES(@u-id, @app-id)
WHERE exists(SELECT app-id, dvc-id
             FROM supported_by)

DELETE FROM add_wish
WHERE add_wish.app-id = @app-id
```

Wallet Page

The page below is the UI design for the user's wallet page for our system.

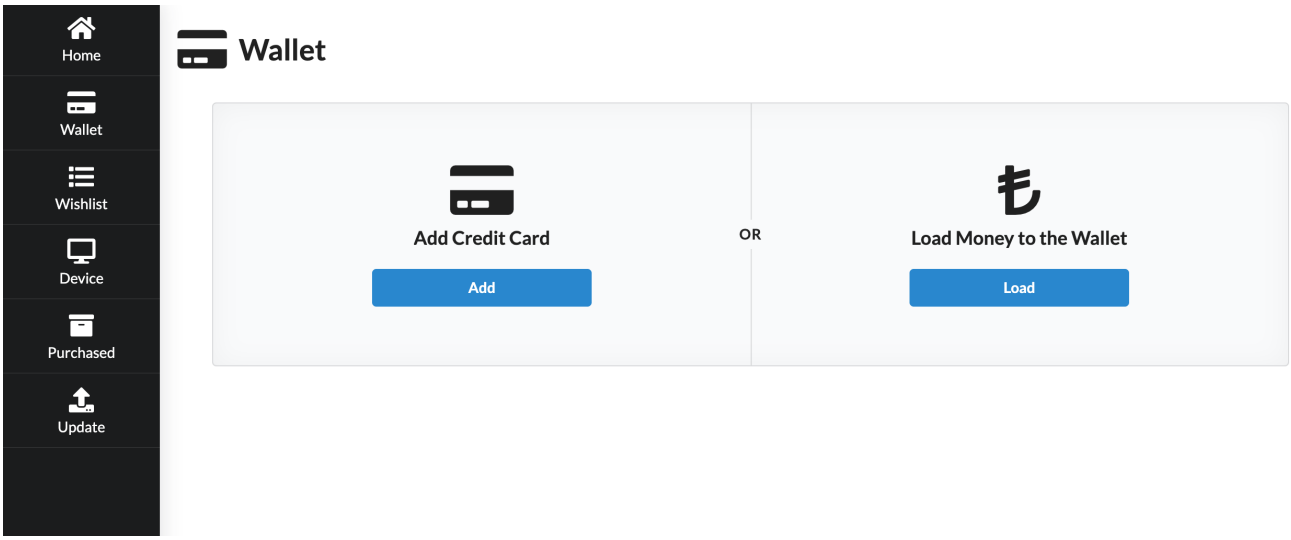

Figure 27: UI design of wallet page

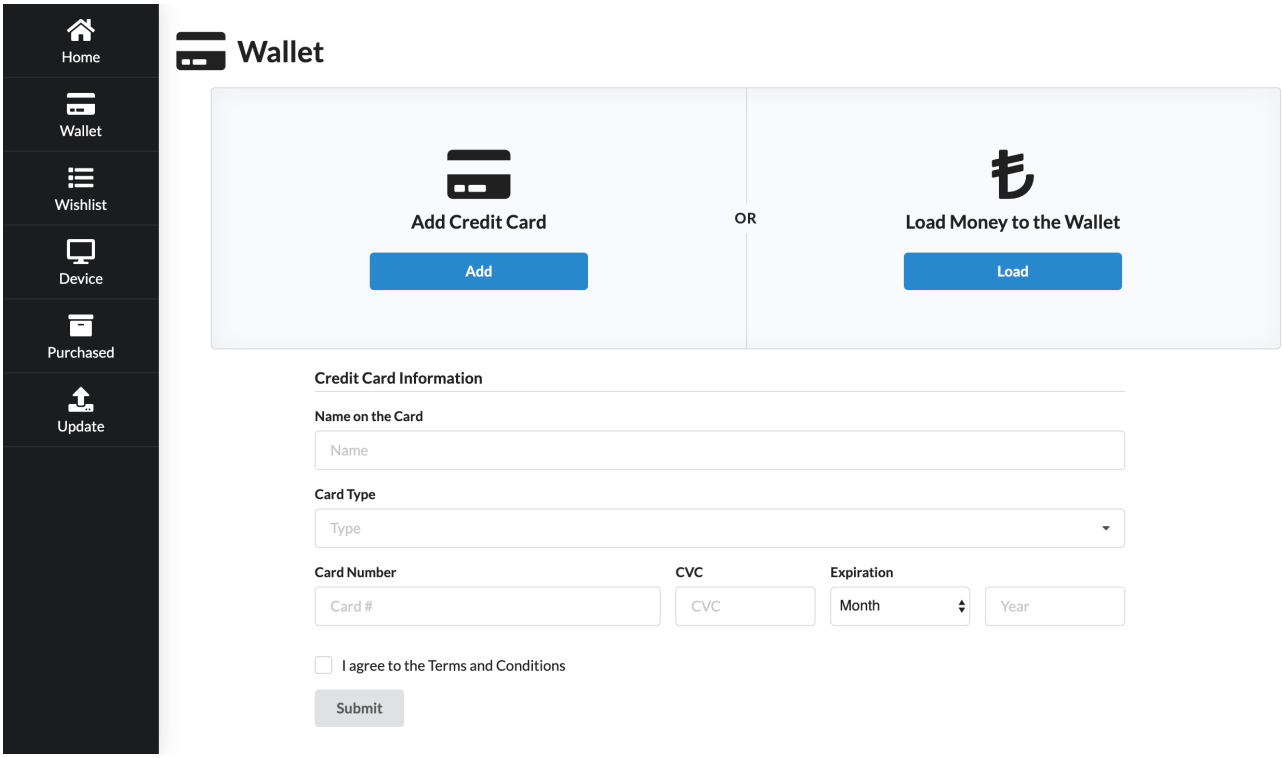

Figure 28: UI design of adding credit card in wallet page

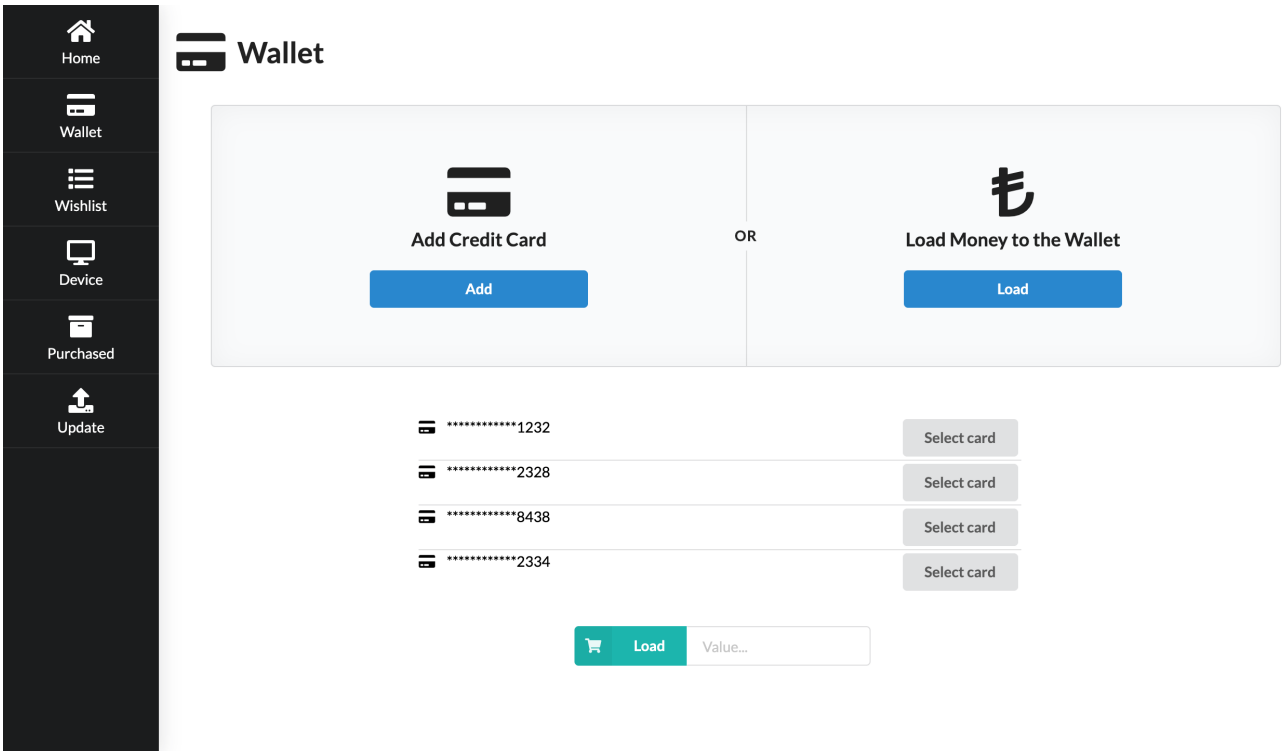

Figure 29: UI design of adding loading money in wallet page

A user can add credit cards into the system by using Wallet page. After clicking add credit card button, we will take Card Type as @bank_name, Card Number as @card_no, expiration date as @exp_date, security number(CVC) as sec_no with encryption, and users Wallet id as @w_id.

Getting User's Wallet ID:

```
SELECT id
FROM Wallet
WHERE u-id = @user_id
```

Add Credit Card:

```
INSERT INTO Card(bank, card-no, exp-date,@sec_no , w-id)
VALUES(@bank_name, @card_no, @exp_date,     ENCODE(@sec_no,'secret'), @w-id)
```

Load Money to the Wallet: We will get user's selected card information and send it to the proper payment API. If the result is affirmative then we will update the wallet table. We'll take @card_num as selected card number, @amount as the amount of money which will be added into the wallet. The following query will send to the payment API if it is not null:

```
SELECT bank,card-no,exp-date,DECODE('sec_no','secret')
FROM Card as C
WHERE C.card-no = @card_num
```

If the previous process completed successfully, we will update the wallet balance with the following query:

```
UPDATE Wallet(balance)
SET balance = balance + @amount
WHERE Wallet.u-id=@user_id
```

Payment Page

The page below is the UI design for the payment page for our system.

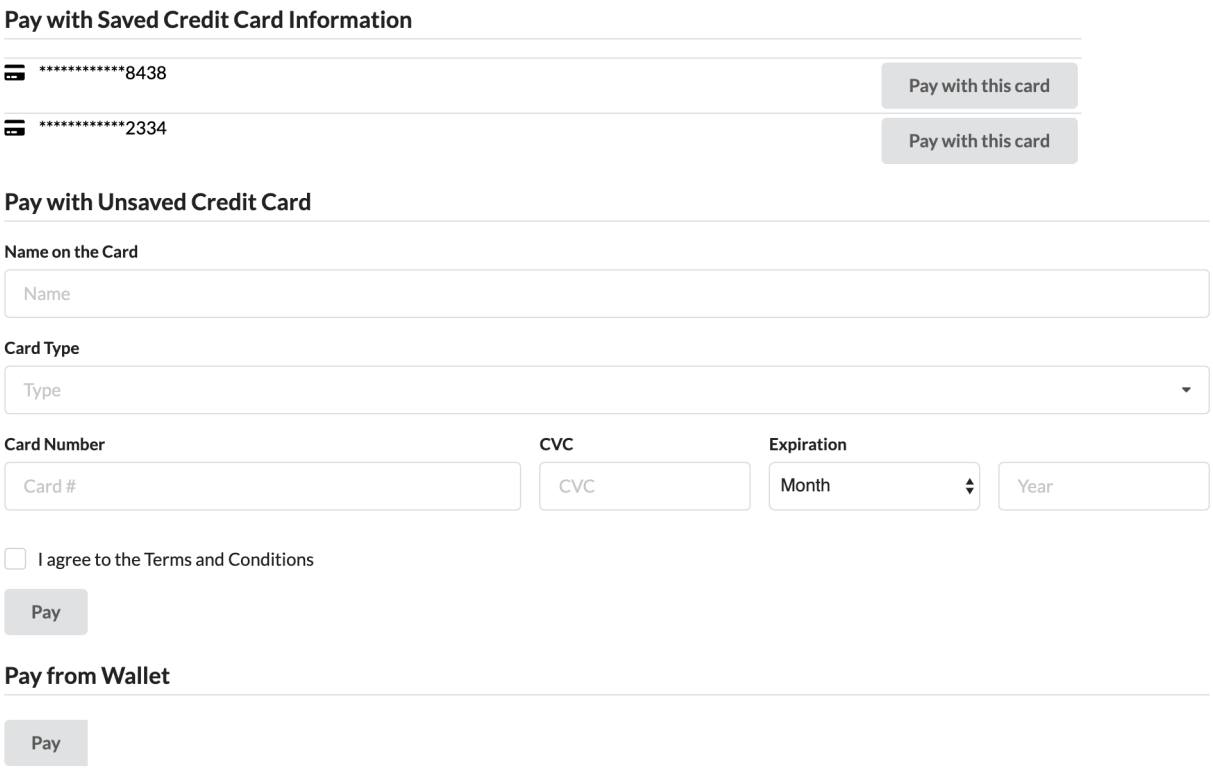

Figure 30: UI design of payment page

Some of the SQL states are same as the ones in the download free app part. These ones will not be repeated to avoid redundancy.

If a user uses his/her balance from the wallet for payment method, the program will check if there is enough amount in the wallet. If user has enough money then, we will decrease the amount from the wallet balance.

Support Control:

```
SELECT *
FROM supported_by
WHERE app-id=@app_id
```

App Price:

```
SELECT price
FROM Application
WHERE id = @app_id
```

Wallet Balance:

```
SELECT balance
FROM Wallet
WHERE u-id=@user_id
```

Decreasing Wallet Account:

```
UPDATE Wallet
SET balance= @balance-@price
WHERE u-id=@user_id
```

Successful Purchase:

```
INSERT INTO owns(u-id,app-id)
VALUES(@user_id, @app_id)
```

Wishlist Page

The page below is the UI design for the user's wishlist page for our system.

Figure 31: UI design of wishlist page

Add Apps into the Wishlist: A user can add any application into his/her wish list. We'll take @user_id as id of user, @app_id as id of wished app.

```
INSERT INTO add_wish(u-id, app-id)
VALUES(@user_id,@app_id)
```

Remove Apps from the Wishlist: A user can remove application from his/her wish list.

```
DELETE FROM add_wish
WHERE EXISTS
(SELECT * FROM add_wish WHERE app-id=@app_id)
```

Display Apps in the Wishlist:

```
SELECT *
FROM add_wish
WHERE u-id = @user_id
```

Update Page

The page below is the UI design for the user's update page for our system. The User will be able to see updated for all of the applications that they have downloaded to the device they are using. When user updates one of their application, this page will be also be updated.

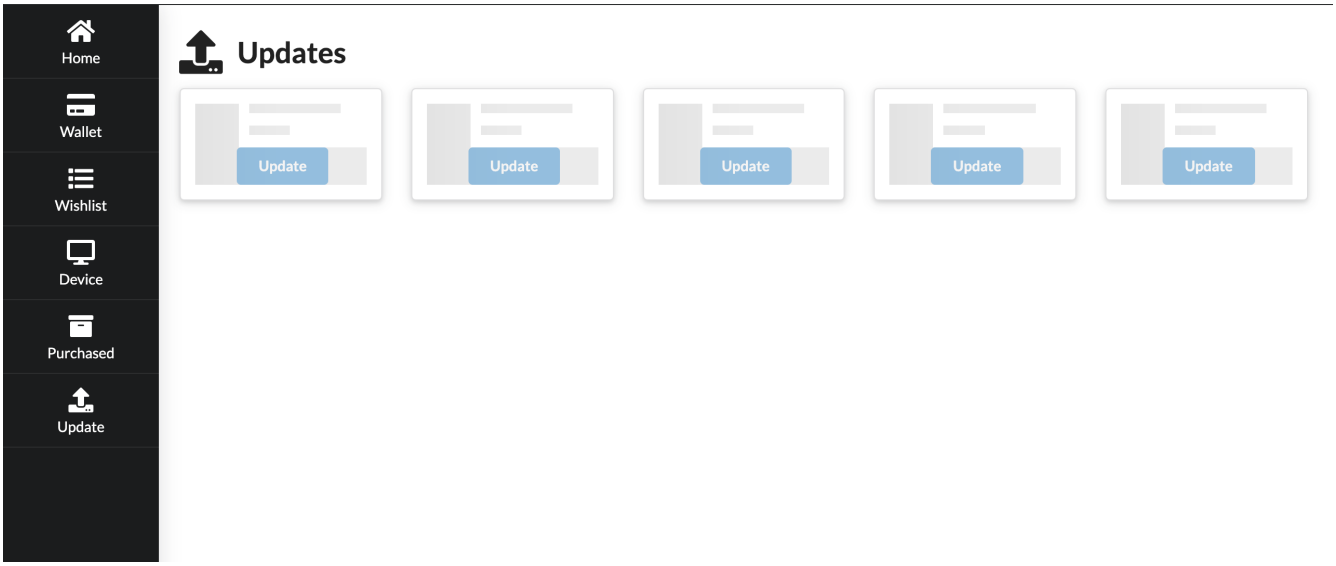

Figure 32: UI design of update page

Display all of the users applications that require update:

```
SELECT A.name, A.description
FROM Application A, owns O
WHERE A.id = O.app-id
AND O.user-version < A.version
```

After applications are updated:

```
UPDATE owns
SET owns.user_version = Application.version
FROM owns, Application
WHERE owns.app-id = Application.id
```

Purchased Page

The page below is the UI design for the user's purchased applications for our system.

Figure 33: UI design of purchased page

A user can buy an app via credit card or wallet. We'll take id of application as @app_id, price of app as @app_price, user balance as @u_balance.At first we check weather the user's device supports the application or not. If user balance is greater than or equal to the price of application, then we can update the Wallet table and add the application into owns table. If the Wallet balance is smaller than price then the system will give a warning.

Display Purchased Apps:

```
SELECT *
FROM owns, Application
WHERE u-id=@user_id AND owns.app-id=Application.id
```

Devices Page

The page below is the UI design for the user's devices page for our system.

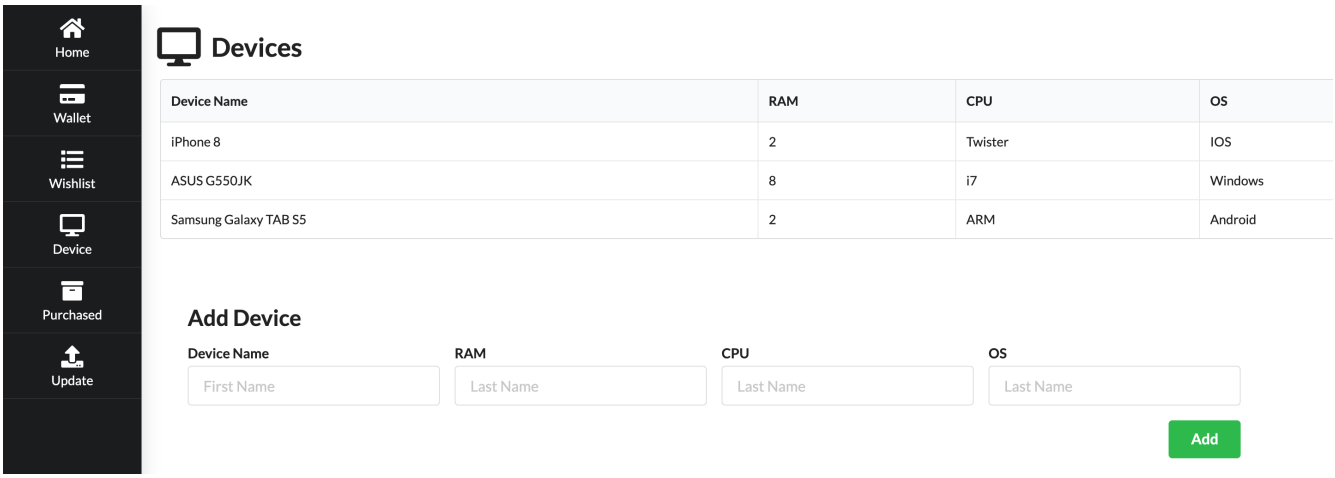

Figure 34: UI design of devices page

When a user first enters the system or owns a new device s/he has to submit new device into the system. We will take device id as @dev_id, device ram as @dev_ram, device cpu as @dev_cpu, device operating system as @dev_os, model of device as @dev_model, name of device as @dev_name, user id number as @user_id. Adding a new device into system query is the following.

```
SELECT *
FROM Device
WHERE ram = @dev_ram AND cpu=@dev_cpu, AND os=@dev_os AND model=@dev_model
```

If the above query returns a empty table then this means that the device is about to add into system is a new device. We can proceed to insertion the new device into Device table as following:

Add a Device into Device table:

```
INSERT INTO  Device(id,ram, cpu, os, model)
VALUES(@dev_id,@dev_ram, @dev_cpu, @dev_os, @dev_model)
```

When a user wants to add a device, we will first find the dev-id from the model of the device and second we want a name for device from user. We'll take model of device as @dev_model, name of device as @dev_name.

Find dev-id:

```
SELECT id as dev_id
FROM Device
WHERE model = @dev_model
```

Add Device into possess table:

```
INSERT INTO  possess(u-id, dev-id, name)
VALUES(@user_id, @dev_id, @dev_name)
```

To display all the devices a user has the following query will be executed.

Display Possessed Devices:

```
SELECT *
FROM possess
WHERE u-id = @user_id
```

Developer's My Applications Page

The page below is the UI design for the developer's home page for our system.

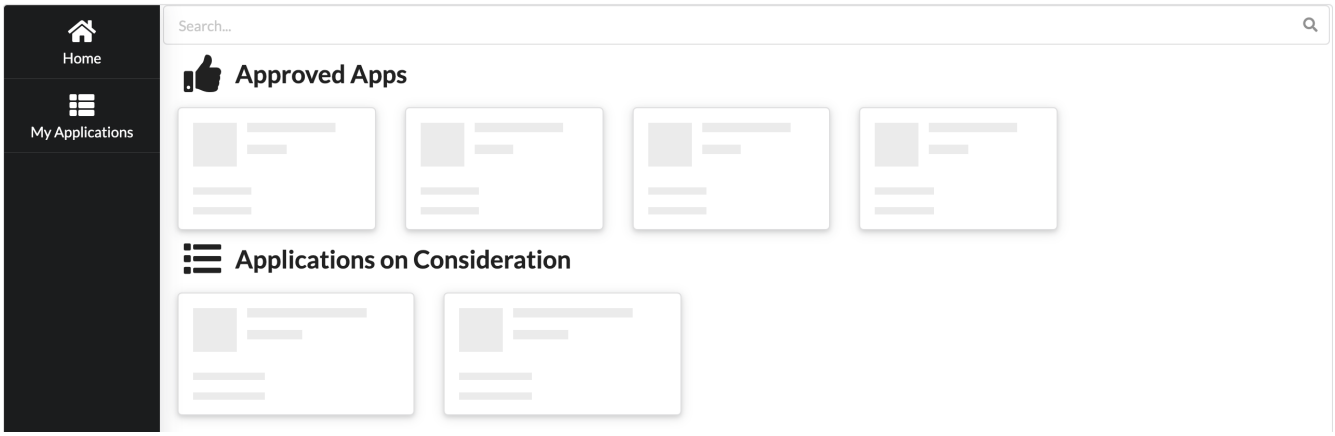

Figure 35: UI design of developer's applications page

On this page, developer can see his/her own apps. There will be two types of apps in this page, the ones approved by an editor and the ones that are waiting to be approved by an editor. We display these two types with the SQL statemets below.

Display Approved Apps:

```
SELECT *
FROM Application A
where id in (SELECT app-id
             FROM Request
             WHERE dev-id = @dev_id
             and state="approved");
```

Display Application on Consideration:

```
SELECT *
FROM Application
where id in (SELECT app-id
             FROM Request
             WHERE dev-id = @dev_id
             and state="pending");
```

Developers will be able to upload an application or an updated version of an application via this page.

Developer Upload/Update Application Page

The page below is the UI design for the upload and update application page which will be accessible only to developers. The developer will fill the form with relevant information and will either click on Upload or Update. The information given by the developer will be locally kept in @name, @description, @size, @price and @category_name. Both publishing a new application and issuing an update involves adding a request to the system.

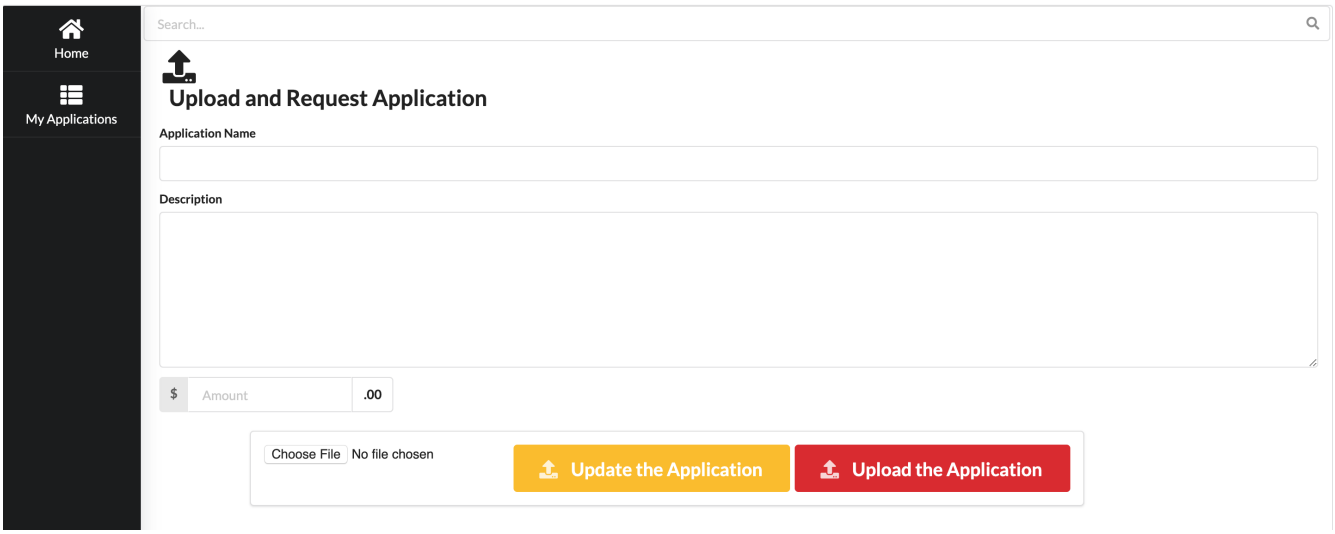

Figure 36: UI design of add application home page

Publish Application:

```
INSERT INTO Application(name, description, size, price, dl_count, version,
↪  dev-id, date)
VALUES (@name, @description, @size, @price, 0, 1, NOW());

INSERT INTO belong_to(c-name, app-id)
VALUES(@category_name,@app_id);

INSERT INTO Request(type, app-id, dev-id, e-id, state)
```

```
SELECT "publish", Application.id, @dev_id, NULL, "pending"
FROM Application WHERE name = @name and description = @description and size =
↪  @size and price = @price and dl_count = 0 and version = 1 and dev-id =
↪  @dev_id;
```

Update Application:

```
UPDATE Application
SET description = @description, version = @version
WHERE app-id = @app-id;

INSERT INTO Request(type, app-id, dev-id, e-id, state) VALUES "update",
↪  @app_id, @dev_id, NULL, "pending";
```

Editor Home Page

The page below is the UI design for the editor's home page for our system. In the home page the Editors will see all of the requests that are waiting for an answer. They will be able to directly reject, accept with restriction or directly accept the request.

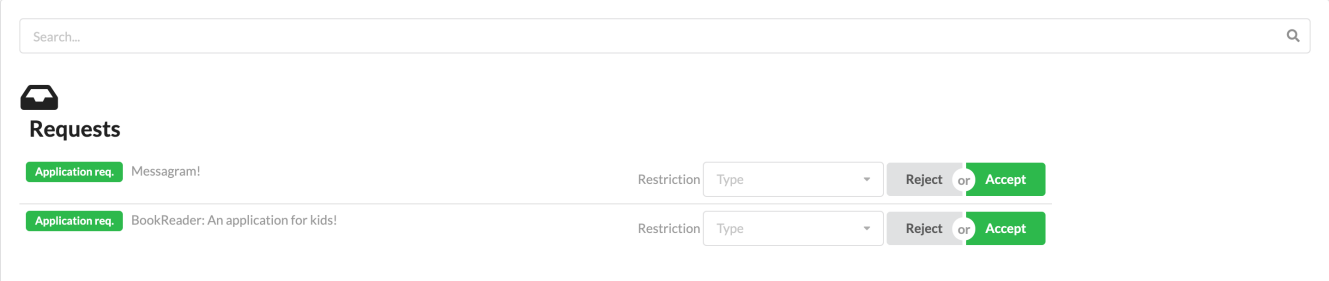

Figure 37: UI design of editor home page

Display All Requests:

```
SELECT type, app-id, dev-id
FROM Request
WHERE state='pending';
```

Editor Rejected a Request:

```
UPDATE Request
SET state = 'rejected'
where id = @req_id
```

Editor Accepted a Request with Restrictions:

```
UPDATE Request
SET state = 'approved', e-id = @editor_id
where id = @req_id;

INSERT INTO Restriction(type, e-id)
VALUES @restriction, @editor_id;

INSERT INTO contains(r-id, app-id)
SELECT @restriction, Request.app-id
FROM Request
WHERE req-id = @req_id
```

Admin Home Page

The page below is the UI design for the admin's home page for our system.

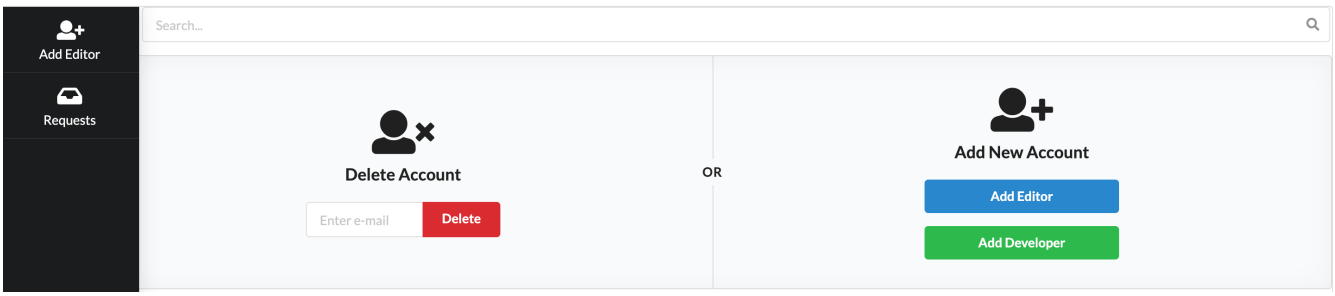

Figure 38: UI design of admin home page

For deletion account operations in admin, the only thing we need is an email address. We will find the id of the account via following query:

```
SELECT id
FROM Account
WHERE email=@email
```

We will find the type of user from the account id and perform the following queries according to user's account type.

Delete User: The query for deleting a user is the following:

```
DELETE FROM Account
WHERE id = @user_id
```

Delete Editor: The query for deleting a editor is the following:

```
INSERT INTO rm_ed(e-id, adm-id)
VALUES(@editor_id,@admin_id)

DELETE FROM Account
WHERE id = @editor_id
```

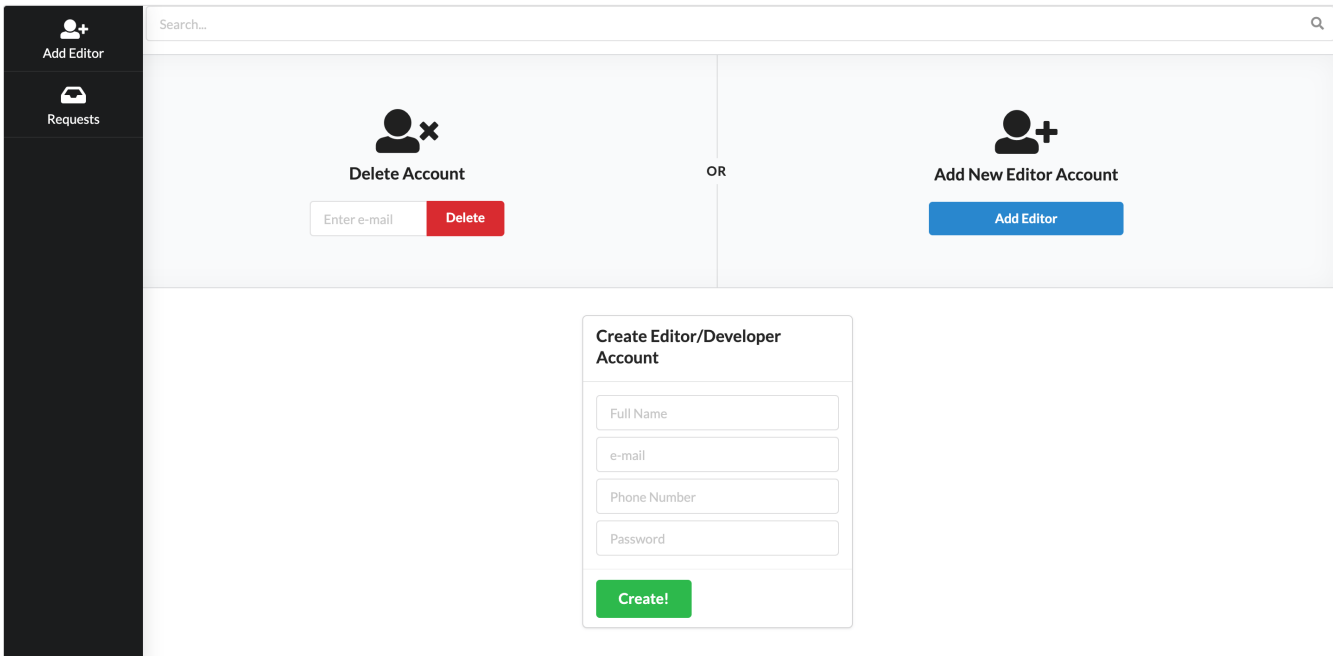

Figure 39: UI design of admin creating editor or developer accounts

Add Editor: An admin can add editors. We'll take @admin_id as admin id and @editor_id as editor id.

```
INSERT INTO Editor(id, adm-id)
VALUES(@editor_id,@admin_id)
```

Add Developer: An admin can add developers. We'll take @admin_id as admin id and @dev_id as developer id.

```
INSERT INTO Developer(id, adm-id)
VALUES(@dev_id,@admin_id)
```

Super Admin/Admin Request Page

The page below is the UI design for the admin's home page for our system.

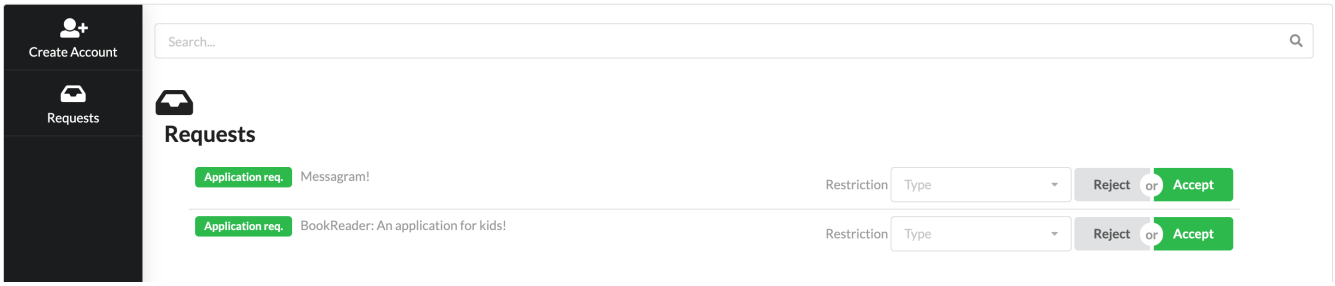

Figure 40: UI design of super(admin) request page

The SQL statements for this UI is given under the mock up for Editoe homa page(Figure 37)The only difference in the statement is the replacement of @e-id with @admin-id and @sadmin-s

Super Admin Home Page

The page below is the UI design for the super admin's home page for our system.

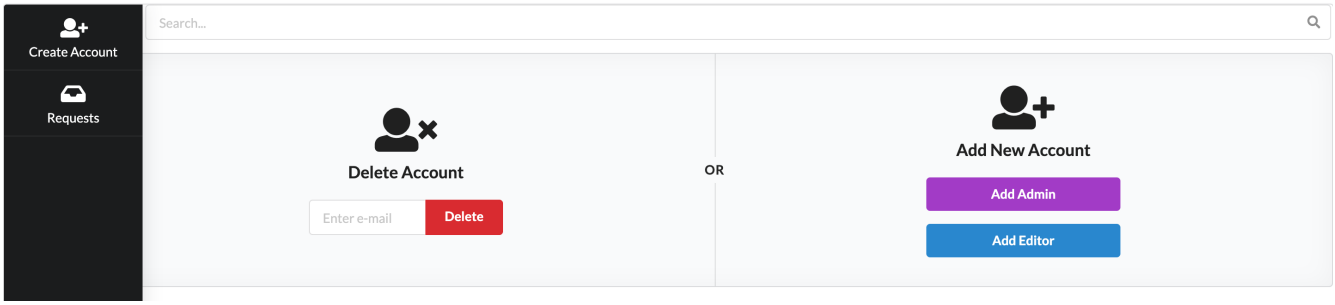

Figure 41: UI design of super admin home page

For deletion account operations in super admin, the only thing we need is an email address. We will find the id of the account via following query:

```
SELECT id
FROM Account
WHERE email=@email
```

We will find the type of user from the account id and perform the following queries according to user's account type.

Delete User: Since a user may has records in, user, premium user, owns, possess,card and wallet tables, we have to delete them all. We used on delete cascade when we creating tables which has foreign key as user-id, to make it easy to delete all records of a deleted user. We'll take id of user as @user_id. The query is the following:

```
DELETE FROM Account
WHERE id = @user_id
```

Delete Editor: An editor may has records in request and restriction tables. We will set e-id as null in request and restrictions tables. We used on delete set null when creating request and restrictions tables. We will add this deletion record into rm_ed table. We'll take @editor_id as removed editor id and @s_id as super admin id who removed editor.The query is the following:

```
INSERT INTO rm_ed(e-id, adm-id)
VALUES(@editor_id,@s_id)


DELETE FROM Account
WHERE id = @editor_id
```

Delete Admin: An admin may has records editor table. We will set e-id as null in editor table. We used on delete set null when creating editor table. We will add this deletion record in rm_ad table. We'll take @admin_id as removed admin and @s_id as super admin who removed the admin. The query is the following:

```
INSERT INTO rm_ad(adm-id, s-id)
VALUES(@admin_,d,@s_id)
```

```
DELETE FROM Account
WHERE id = @admin_id
```

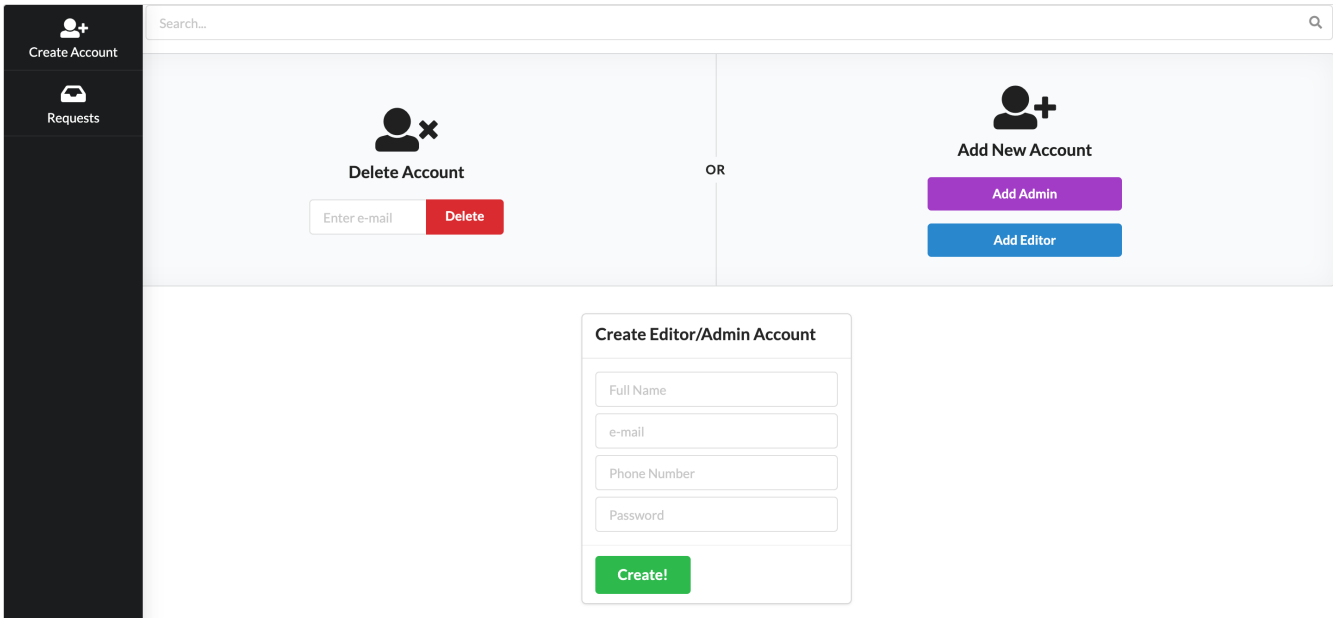

Figure 42: UI design of super admin creating accounts

Add Admin: A super admin can add admins. We'll take @s_id as super admin id and @admin_id as admin id.

```
INSERT INTO Admin(id, s-id)
VALUES(@admin_id,@s_id)
```

Add Editor: A super admin can add editors. We'll take @s_id as super admin id and @editor_id as editor id.

```
INSERT INTO Editor(id, adm-id)
VALUES(@editor_id,@s_id)
```

Add Developer: A super admin can add developers. We'll take @s_id as super admin id and @dev_id as developer id.

```
INSERT INTO Developer(id, adm-id)
VALUES(@dev_id,@s_id)
```

6 Advanced Database Components

6.1 Views

Get Approved Applications

```
CREATE VIEW approved_apps AS
SELECT A.id, A.name, A.description, A.size, A.price, A.dl_count, A.version,
↪  A.dev-id, A.date
FROM Request R , Application A
WHERE R.app-id = A.id AND R.state = 'approved'
```

6.2 Reports

Retrieve the Applications with the Highest Average Rate for Each Category

```
WITH average (app-id, avg-rate) as(
    SELECT app-id, avg(rate)
    FROM Review
    GROUP BY app-id
    )
WITH best-avg-rate(c-name, max-rate) as(
    SELECT B.c-name, max(avg-rate)
    FROM Application A, average Av, belong_to B
    WHERE Av.app-id = A.id AND B.app-id = A.id
    GROUP BY B.c-name
    )
SELECT B.c-name, A.app-id
FROM Application A, belong_to B, best-avg-rate Best, Review R
WHERE A.id = B.app-id AND B.c-name = Best.c-name AND R.app-id = A.id AND
↪  R.rate = Best.max-rate
```

Retrieve the Number of Applications Purchased and in the Wishlist Along with the Number of Devices for Each User

```
SELECT O.u-id , count( W.app-id), count( P.dvc-id), count( O.app-id)
FROM add_wish W, possess P, owns O
WHERE W.u-id = P.u-id AND P.u-id = O.u-id
GROUP BY O.u-id
```

Retrieve Users and Their Comments Who Have Downloaded "Angry Birds" and Rated Below 2

```
SELECT O.u-id, R.rate, R.comment
FROM owns O, Application A, Review R
WHERE A.id = O.app-id AND A.name = 'Angry Birds' AND R.rate < 2
```

Retrieve the Amount of Money that Each Developer Gained

```
SELECT D.dev-id, Ac.name, dl_count x price
FROM  Application A, Developer D, Account Ac
WHERE Ac.id = D.id AND A.dev-id = D.id
```

6.3 Triggers

Checking that an Added Restriction Has Type 'G', 'PG', 'PG-13', 'R', 'NC-17'

```
DELIMITER //
CREATE TRIGGER rest_type_check AFTER green INSERT ON Restriction
REFERENCING NEW ROW AS nrow
FOR EACH ROW
WHEN(nrow.type not in {'G', 'PG', 'PG-13', 'R', 'NC-17'})
BEGIN
ROLLBACK END //
DELIMITER ;
```

6.4 Constraints

The format of integrity constraints of creation of tables are the followings:

- primary key(A1,....,An)
- foreign key(Ax,.....An) references r
- not null
- unique

Thus, renaming is unnecessary.

6.5 Stored Procedure

DELIMETER//

```
CREATE PROCEDURE get_avg_rating_of_app(IN app_id INT, OUT avg_rating NUMERIC
↪  (1,1))
BEGIN
    SELECT AVG(rate) INTO avg_rating
    FROM Review as R
    WHERE  R.app-id= app_id;
END //
DELIMETER ;
```

7 Implementation Plan

For front-end implementation we used Semanctic UI framework. Improvements, will be done by using bootstrap and Semantic UI. For back-end implementation, we will use Javascript and PHP. For database implementation, we will be using MySQL.

8 Website

Our website is: https://babademez.github.io/Zebra/

References

[1] A. Silberschatz, H. F. Korth, and S. Sudarshan, *Database system concepts*, 6th ed. New York: McGraw-Hill, 2010. [Online]. Available: http://www.db-book.com/

[2] R. Elmasri and S. Navathe, *Fundamentals of database systems.* Addison-Wesley Reading, Mass, 2000.