Bilkent University

Department of Computer Engineering

Agora

A Social Discussion Website Database System

Group 8

Nashiha Ahmed | 21402950

Raza Faraz |21404239

Cholpon Mambetova |21402612

Selin Özdaş |21400537

Professor Özgür Ulusoy

CS353 Database Systems Course | Section

Due: Nov 20, 2017

Design Report

Contents

1| Introduction

This report is submitted as a project proposal as part of the CS 353 Database Systems term project. Agora is present is a social discussion website database resembling Reddit, Quora, or StackOverflow. The report details solicited description of the term project, the functional and nonfunctional requirements, limitations, and Entity-Relationship model of the project, respectively.
Progress of the project can be followed through https://github.com/selinozdas/CS353.

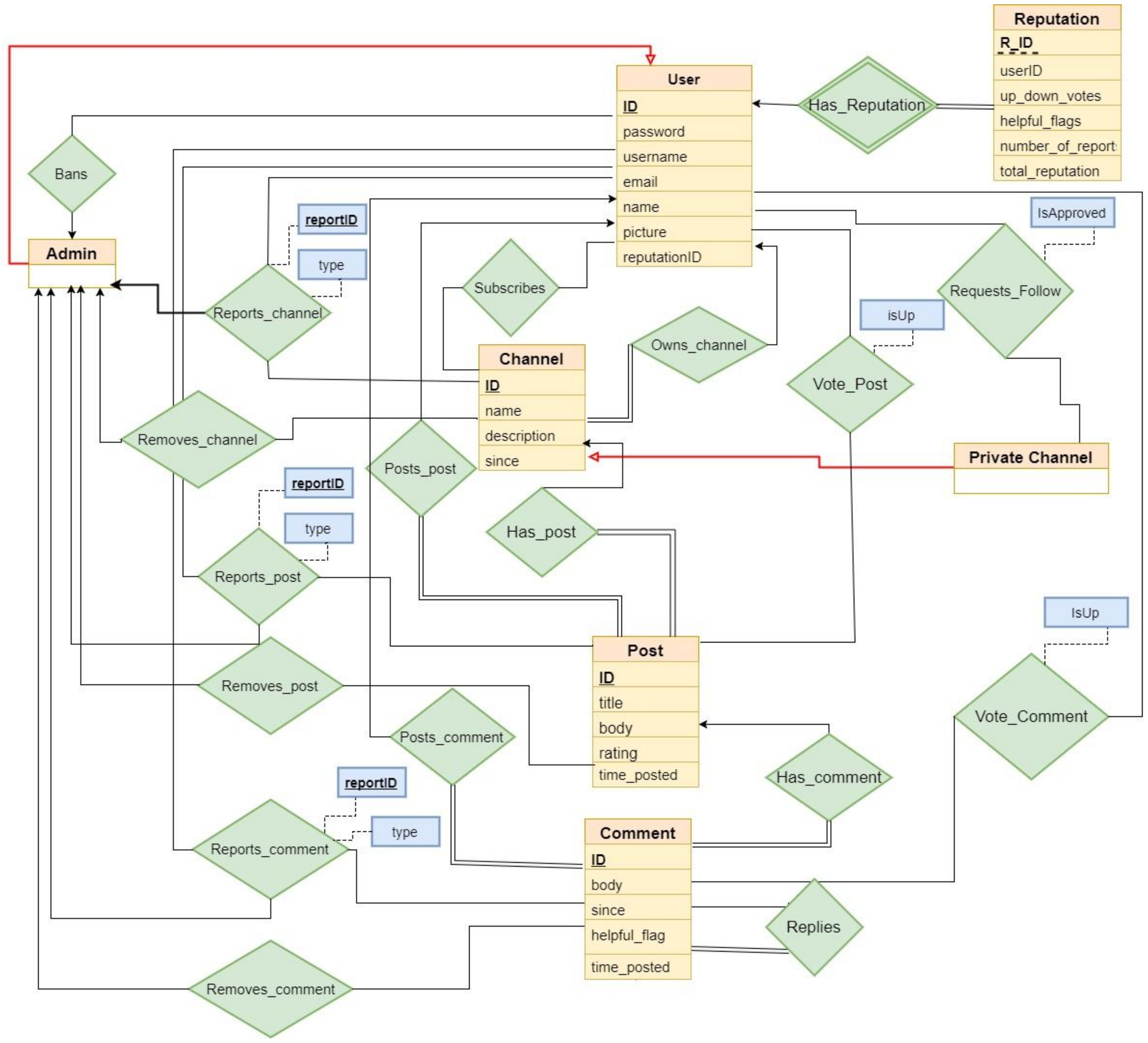
2| Project Description

Agora is a social discussion website that can be used to discuss subjects such as news events, politics, distinct skills knowledge, inquiries, academics, interests, or personal life. Agora has the potential to bring users together who want to ask, learn, share or discuss a variety of subjects. Our system will be a collection of individual knowledge. Visitors can view Agora forums, but these visitors will not be entitled to certain privileges pertaining to registered users. These privileges include subscribing to channels, posting, commenting, rating, and reporting forums or comments. Registered users must have accounts with passwords and unique usernames.

Since social discussion applications (e.g. Reddit, Stackoverflow or Quora) consist of many entities and relations, there is copious information needed to be stored and managed. Database management systems are suitable for these social discussion applications, due to their provision of a highly efficient method to handle multiple types of sizable data. Another motivation to use a database management system is the dynamic data flow in our system. Agora must be quickly accessible and easily-updatable in case of changes or extensions. A database management system is suitable for the defined requirements in section 3 of the report. For example, our system must be accurate and secure and must avoid data redundancy which can be achieved with an efficient database design. Therefore, Agora will employ a database management system [1].

3| Agora Entity-Relationship Model

3.1| (Revised) Entity-Relationship Diagram



This entity relationship model displays revised core relationships and entities after feedback on the previous E/R model was given. Each will be explained in further detail on the following page.

3.2| (Revised) Entity-Relationship Diagram Description

- Instead of having composite "reputation" attribute in "User" entity, we included a "Reputation" relation which includes both voting and helpful flag functionalities along with number of reports that a user received. The "total_reputation" of a user is calculated according to those, and shows the total reputation as one variable to reduce complications. The reputation relation has a weak relation with user, since it is identified by the user relation.
- We changed the cardinalities of the relations with Admin from many-to-many to one-to-many.
- We changed the name of "Forum" to "Post" to be more clearly that we have a "Post" entity.
- For the relations "Reports_comment" and "Reports_post", and "Reports_channel" we made reportID a primary key.
- We changed the name of the closed channel to "private" channel.
- Before a private channel had moderators and followers. To reduce redundancy and duplication we removed it
- For clarity we have changed the name of the entity "Send_request" to "Requests_follow" as we felt that a user is actually requesting to follow that channel. We also changed the name of attribute "Approved" to "isApproved" to clearly state the type of an attribute which is boolean.
- We included three tertiary relationships. These are the report relationships: "Report_channel", "report_comment", and "report_post." They are between user, admin, and either channel, comment, or post respectively.
- We added voting relations as "Vote_Post" and "Vote_comment". Both of them have a boolean variable "isUp", meaning if user wants to upvote, the variable will be true, if user wants to downvote, variable will be false.
- We renamed "Registered-User" to "User" to reduce unnecessary wording.
- We added IDs to "User" and "Channel" to improve performance of the system, especially searching functionalities.
- We added "time_posted" attributes to entities "Post" and "Comment", to have an access to time those were posted; and therefore, make our system more complete.
- We changed "Posts_post" and "Posts_comment" participation from partial to total.

3.3 | Relational Schemas

1. User

Relational Model:

User (ID, username, password, email, name, picture, reputationID)

Functional Dependencies:

ID $\rightarrow$ username, password, email, name, picture, reputationID,
username $\rightarrow$ ID, password, email, name, picture, reputationID,
email $\rightarrow$ ID, username, password, name, picture, reputationID,

Candidate Keys:

{(ID),(username), (email)}

Normal Form:

3NF

Table Definition:

```
CREATE TABLE User(
        ID INT (10) PRIMARY KEY AUTO_INCREMENT
        username VARCHAR (20) UNIQUE NOT NULL,
        password VARCHAR (20) NOT NULL,
        email VARCHAR (50) UNIQUE NOT NULL,
        name VARCHAR (50) NOT NULL,
        picture VARCHAR(100),
        reputationID INT (10),
        helpful_flags  INT DEFAULT 0);
```

2. Admin

Relational Model:

Admin (<u>ID</u>)

Functional Dependencies:

None

Candidate Keys:

{(ID)}

Normal Form:

3NF

Table Definition:

```
CREATE TABLE User(
        ID INT (6) PRIMARY KEY,
        FOREIGN KEY (ID) REFERENCES User (ID));
```

3. Channel

Relational Model:

Channel (<u>ID</u>, name, description, since)

Functional Dependencies:

ID $\rightarrow$ name, description, since
name $\rightarrow$ ID, description, since

Candidate Keys:

{(ID), (name)}

Normal Form:

3NF

Table Definition:

```
CREATE TABLE Channel(
        ID INT (10) UNSIGNED PRIMARY KEY AUTO_INCREMENT,
        name VARCHAR (32) NOT NULL,
        description VARCHAR (800),
        since TIMESTAMP);
```

4. Private Channel

Relational Model:

PrivateChannel (ID)

Functional Dependencies:

None

Candidate Keys:

{(ID)}

Normal Form:

3NF

Table Definition:

```
CREATE TABLE PrivateChannel(
        ID INT (10) PRIMARY KEY,
        FOREIGN KEY (ID) REFERENCES Channel (ID));
```

5. Post

Relational Model:

Post (ID, title, body, rating, time_posted)

Functional Dependencies:

ID $\rightarrow$ title, body, rating, time_posted

Candidate Keys:

{(ID)}

Normal Form:

3NF

Table Definition:

```
CREATE TABLE Post(
        ID INT(10) UNSIGNED PRIMARY KEY AUTO_INCREMENT
        title VARCHAR (100) NOT NULL,
        body VARCHAR NOT NULL,
        rating INT DEFAULT 0,
        time_posted TIMESTAMP);
```

6. Comment

Relational Model:

Comment (<u>ID</u>, body, since, helpful_flags, time_posted)

Functional Dependencies:

ID $\rightarrow$ body, since, helpful_flags, time_posted

Candidate Keys:

{(ID)}

Normal Form:

3NF

Table Definition:

```
CREATE TABLE Comment(
        ID INT(10) UNSIGNED PRIMARY KEY AUTO_INCREMENT
        body VARCHAR NOT NULL,
        since TIMESTAMP NOT NULL,
        rating INT DEFAULT 0
        helpful_flags INT UNSIGNED DEFAULT 0,
        time_posted TIMESTAMP);
```

7. Reputation

Relational Model:

Reputation (R_ID, userID, up_down_votes, helpful_flags, number_of_reports, total_reputation)

Functional Dependencies:

R_ID $\rightarrow$ userID, up_down_votes, helpful_flags, number_of_reports, total_reputation
up_down_votes, helpful_flags, number_of_reports $\rightarrow$ total_reputation

Candidate Keys:

{(R_ID)}

Normal Form:

3NF

Table Definition:

```
CREATE TABLE Reputation(
        R_ID INT (10) PRIMARY KEY AUTO_INCREMENT,
        userID INT (10) NOT NULL,
        up_down_votes INT DEFAULT 0,
        helpful_fags INT DEFAULT 0,
        number_of_reports INT DEFAULT 0,
        total_reputation INT DEFAULT 0,
        FOREIGN KEY (userID) REFERENCES User (ID));
```

8. Subscribes

Relational Model:

Subscribes (<u>userID</u>, <u>channelID</u>)

Functional Dependencies:

None

Candidate Keys:

{(userID, channelID)}

Normal Form:

3NF

Table Definition:

```
CREATE TABLE Subscribes(
        userID INT (10),
        channelID INT (10),
        PRIMARY KEY (userID, channelID),
        FOREIGN KEY (userID) REFERENCES User (ID),
        FOREIGN KEY (channelID) REFERENCES Channel (ID));
```

9. Requests To Follow

Relational Model:

Requests_Follow (<u>userID</u>, <u>channelID</u>, isApproved)

Functional Dependencies:

userID, channelID $\rightarrow$ isApproved

Candidate Keys:

{(userID, channelID)}

Normal Form:

3NF

Table Definition:

```
CREATE TABLE Requests_Follow(
        userID INT (10),
        channelID INT (10),
        isApproved BOOLEAN DEFAULT ‘false’,
        PRIMARY KEY (userID, channelID),
        FOREIGN KEY (userID) REFERENCES User (ID),
        FOREIGN KEY (channelID) REFERENCES Channel (ID));
```

10. Owns Channel

Relational Model:

Owns_Channel (<u>userID</u>, <u>channelID</u>)

Functional Dependencies:

None

Candidate Keys:

{(userID, channelID)}

Normal Form:

3NF

Table Definition:

```
CREATE TABLE Own_Channel(
    userID INT (10),
    channelID INT (10),
    PRIMARY KEY (userID, channelID),
    FOREIGN KEY (userID) REFERENCES User (ID),
    FOREIGN KEY (channelID) REFERENCES Channel (ID));
```

11. Has Post

Relational Model:

Has_Post (<u>channelID</u>, <u>postID)</u>

Functional Dependencies:

None

Candidate Keys:

{(channelID, postID)}

Normal Form:

3NF

Table Definition:

```
CREATE TABLE Has_Comment(
        channelID INT (10),
        postID INT (10),
        PRIMARY KEY (channelID, postID),
        FOREIGN KEY (channelID) REFERENCES Channel (ID),
        FOREIGN KEY (postID) REFERENCES Post (ID));
```

12. Has Comment

Relational Model:

Has_Comment (postID, commentID)

Functional Dependencies:

None

Candidate Keys:

{(postID, commentID)}

Normal Form:

3NF

Table Definition:

```
CREATE TABLE Has_Comment(
        postID INT (10),
        commentID INT (10),
        PRIMARY KEY (postID, commentID),
        FOREIGN KEY (postID) REFERENCES Post (ID),
        FOREIGN KEY (commentID) REFERENCES Comment (ID));
```

13. Posts Post

Relational Model:

Posts_post(postID, userID)

Functional Dependencies:

None

Candidate Keys:

{(postID, userID)}

Normal Form:

3NF

Table Definition:

```
CREATE TABLE Posts_post(
        postID INT (10),
        userID INT (10),
        PRIMARY KEY (postID, userID),
        FOREIGN KEY (userID) REFERENCES User (ID),
        FOREIGN KEY (postID) REFERENCES Post (ID));
```

14. Posts Comment

Relational Model:

Posts_comment (<u>commentID</u>, user<u>ID)</u>

Functional Dependencies:

None

Candidate Keys:

{(commentID, userID)}

Normal Form:

3NF

Table Definition:

```
CREATE TABLE Posts_comment(
        commentID INT (10),
        userID INT (10),
        PRIMARY KEY (commentID, userID),
        FOREIGN KEY (commentID) REFERENCES Comment (ID),
        FOREIGN KEY (userID) REFERENCES User (ID));
```

15. Replies

Relational Model:

Replies (parentCommentID, childCommentID)

Functional Dependencies:

None

Candidate Keys:

{(parentCommentID, childCommentID)}

Normal Form:

3NF

Table Definition:

```
CREATE TABLE Replies(
        parentCommentID INT (10),
        childCommentID INT (10),
        PRIMARY KEY (parentCommentID, childCommentID),
        FOREIGN KEY (parentCommentID) REFERENCES Comment (ID),
        FOREIGN KEY (childCommentID) REFERENCES Comment (ID));
```

16. Reports Channel

Relational Model:

Reports_Channels (<u>userID</u>, <u>channelID</u>, <u>reportID</u>, type)

Functional Dependencies:

userID, channelID, reportID $\rightarrow$ type

Candidate Keys:

{(userID, channelID, reportID)}

Normal Form:

3NF

Table Definition:

```
CREATE TABLE Reports_Channel(
        userID INT (10),
        channelID INT (10),
        reportID INT (10),
        type INT NOT NULL,
        PRIMARY KEY (userID, channelID, reportID),
        FOREIGN KEY (userID) REFERENCES User (ID),
        FOREIGN KEY (channelID) REFERENCES Channel (ID));
```

17. Reports Post

Relational Model:

Reports_Post (<u>userID</u>, <u>postID</u>, <u>reportID</u>, type)

Functional Dependencies:

userID, postID, reportID $\rightarrow$ type

Candidate Keys:

{(userID, postID, reportID)}

Normal Form:

3NF

Table Definition:

```
CREATE TABLE Reports_Post(
        userID INT (10),
        postID INT (10),
        reportID INT (10),
        type INT NOT NULL,
        PRIMARY KEY (userID, postID, reportID),
        FOREIGN KEY (userID) REFERENCES User (ID),
        FOREIGN KEY (postID) REFERENCES Post (ID));
```

18. Reports Comment

Relational Model:

Reports_Comment (<u>userID</u>, <u>commentID</u>, <u>reportID</u>, type)

Functional Dependencies:

userID, commentID, reportID $\rightarrow$ type

Candidate Keys:

{(userID, commentlID, reportID)}

Normal Form:

3NF

Table Definition:

```
CREATE TABLE Report_Comment(
        userID INT (10),
        commentID INT (10),
        reportID INT (10),
        type INT NOT NULL,
        PRIMARY KEY (userID, commentID, reportID),
        FOREIGN KEY (userID) REFERENCES User (ID),
        FOREIGN KEY (commentID) REFERENCES Comment (ID));
```

19. Removes Channel

Relational Model:

Remove_channel (<u>channelID</u>, <u>adminID)</u>

Functional Dependencies:

None

Candidate Keys:

{(channelID, adminID)}

Normal Form:

3NF

Table Definition:

```
CREATE TABLE Remove_channel(
        channelID INT (10),
        adminID INT (10),
        PRIMARY KEY (channelID, adminID),
        FOREIGN KEY (channelID) REFERENCES Channel (ID),
        FOREIGN KEY (adminID) REFERENCES Admin (ID));
```

20. Removes Post

Relational Model:

Remove_post(<u>postlID</u>, <u>adminID)</u>

Functional Dependencies:

None

Candidate Keys:

{(postID, adminID)}

Normal Form:

3NF

Table Definition:

```
CREATE TABLE Remove_post(
        postID INT (10),
        adminID INT (10),
        PRIMARY KEY (channelID, adminID),
        FOREIGN KEY (postID) REFERENCES Post (ID),

        FOREIGN KEY (adminID) REFERENCES Admin (ID));
```

21. Removes Comment

Relational Model:

Remove_comment(<u>commentID</u>, <u>adminID</u>)

Functional Dependencies:

None

Candidate Keys:

{(commentID, adminID)}

Normal Form:

3NF

Table Definition:

```
CREATE TABLE Remove_comment(
        commentID INT (10),
        adminID INT (10),
        PRIMARY KEY (commentID, adminID),
        FOREIGN KEY (commentID) REFERENCES Comment (ID),
        FOREIGN KEY (adminID) REFERENCES Admin (ID));
```

22. Bans

Relational Model:

Ban(<u>bannedUserID</u>, <u>adminID</u>)

Functional Dependencies:

None

Candidate Keys:

{(bannedUserID, adminID)}

Normal Form:

3NF

Table Definition:

```
CREATE TABLE Ban(
        bannedUserID INT (10),
        adminID INT (10),
        PRIMARY KEY (channelID, adminID),
        FOREIGN KEY (bannedUserID) REFERENCES User(ID),
        FOREIGN KEY (adminID) REFERENCES Admin (ID));
```

23. Votes Post (Upvote and Downvote a post)

Relational Model:

Votes_Post (<u>userID</u>, <u>postID</u>, isUp)

Functional Dependencies:

userID, postID $\rightarrow$ isUp

Candidate Keys:

{(userID, postID)}

Normal Form:

3NF

Table Definition:

```
CREATE TABLE Votes_Post(
        userID INT (10),
        postID INT (10),
        isUp BOOLEAN NOT NULL,
        PRIMARY KEY (userID, postID),
        FOREIGN KEY (userID) REFERENCES User (ID),
        FOREIGN KEY (postID) REFERENCES Post (ID));
```

24. Votes Comment (Upvote and Downvote a comment)

Relational Model:

Votes_Comment (userID, commentID, isUp)

Functional Dependencies:

userID, commentId $\rightarrow$ isUp

Candidate Keys:

{(userID, commentID)}

Normal Form:

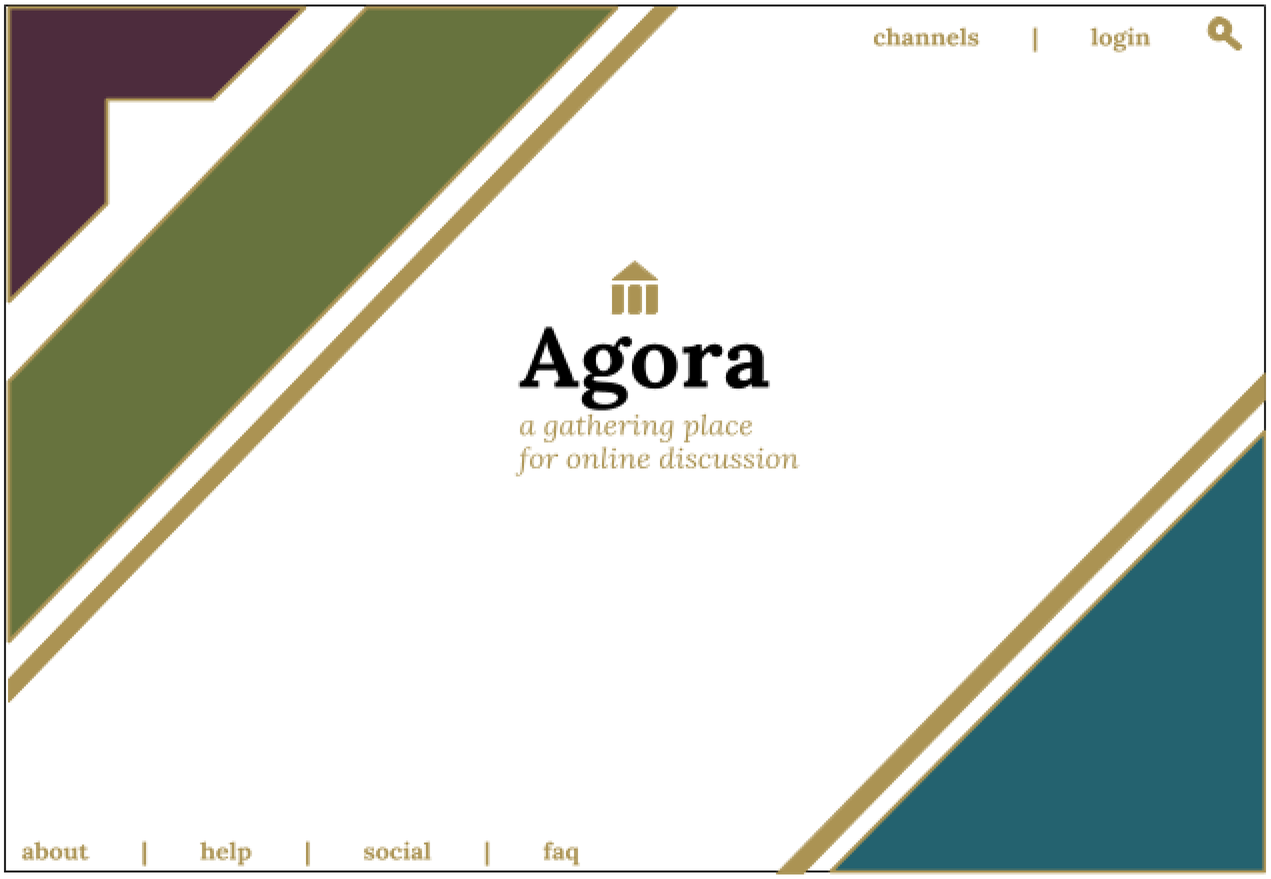
3NF

Table Definition:

```
CREATE TABLE Votes_Comment(
        userID INT (10),
        commentID INT (10),
        isUp BOOLEAN NOT NULL,
        PRIMARY KEY (userID, commentID),
        FOREIGN KEY (userID) REFERENCES User (ID),
        FOREIGN KEY (commentID) REFERENCES Comment (ID));
```

4|User Interface and SQL statements

4.1|Main Page for Non-Registered Users

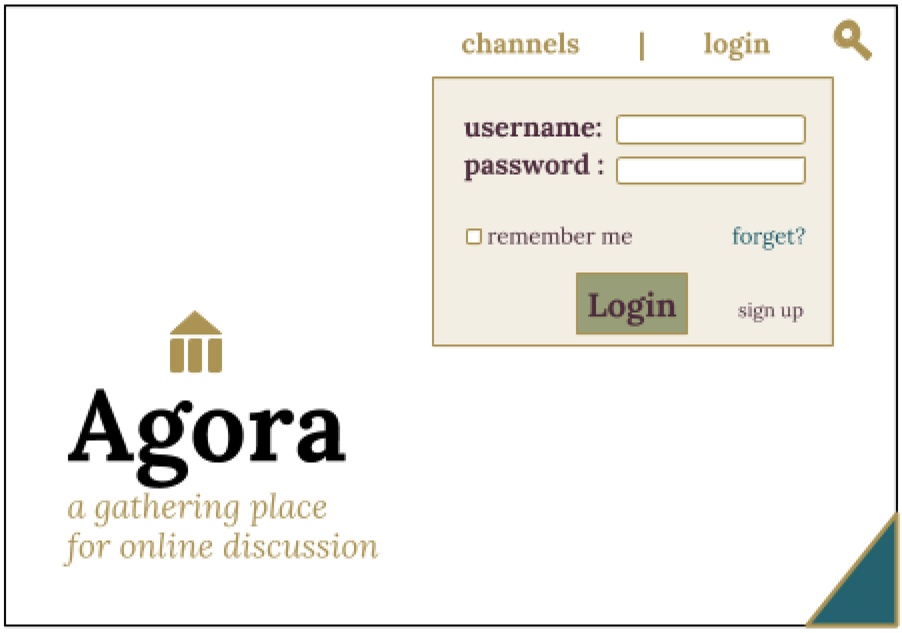


Inputs: None.
Process: The welcoming page of Agora is displayed above. Non-registered users will be greeted by this screen which enables them to login, to see the open channels, and to search queries through Agora simply by clicking on the buttons in the upper right corner. Buttons in the lower left corner will going to lead users to the related pages:

- About : Text based page of information about Agora and their creators
- Help : Text based page of admin contact information
- Social : Links to the other social network channels of Agora
- FAQ (Frequently Asked Questions) : Text based page of frequently asked questions and their answers

SQL statement: None.

4.2| Login Pop-Up



Inputs: @username, @password

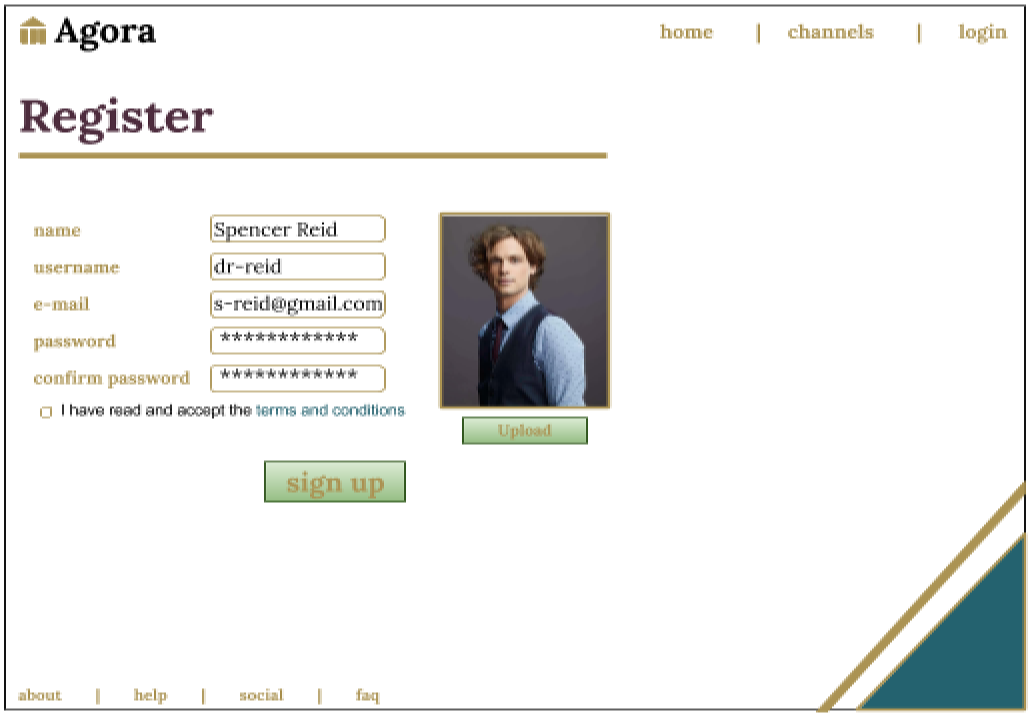
Process: The login pop-up is located to the top right of the main page. They can enter to the system by using their usernames and passwords. The pop-up presents the option to navigate to the lost password screen or registration screen as well. When user is logged in he/she going to see Channels Page

SQL statements:

(retrieve user)

```
select *
from user
where(username=@username and password=@password)
```

4.3| Registration Page



name	Spencer Reid
username	dr-reid
e-mail	s-reid@gmail.com
password	*************
confirm password	*************

☐ I have read and accept the terms and conditions

sign up

Inputs: @username, @password, @confirmpass, @email, @name, @picture_path
Process: The registration page will be directed through the "sign up" button located in the login pop-up. User is expected to fill the form by entering his/her full name, a unique username, e-mail address, password and can upload a profile picture. If the user does not upload any profile picture, logo of Agora will be the default profile picture. Default values for reputation points and helpful flags are going to be set to 0 by default.
SQL statements:
(check if username already exists)

```
select ID from user where username=@username;
```

(create account)

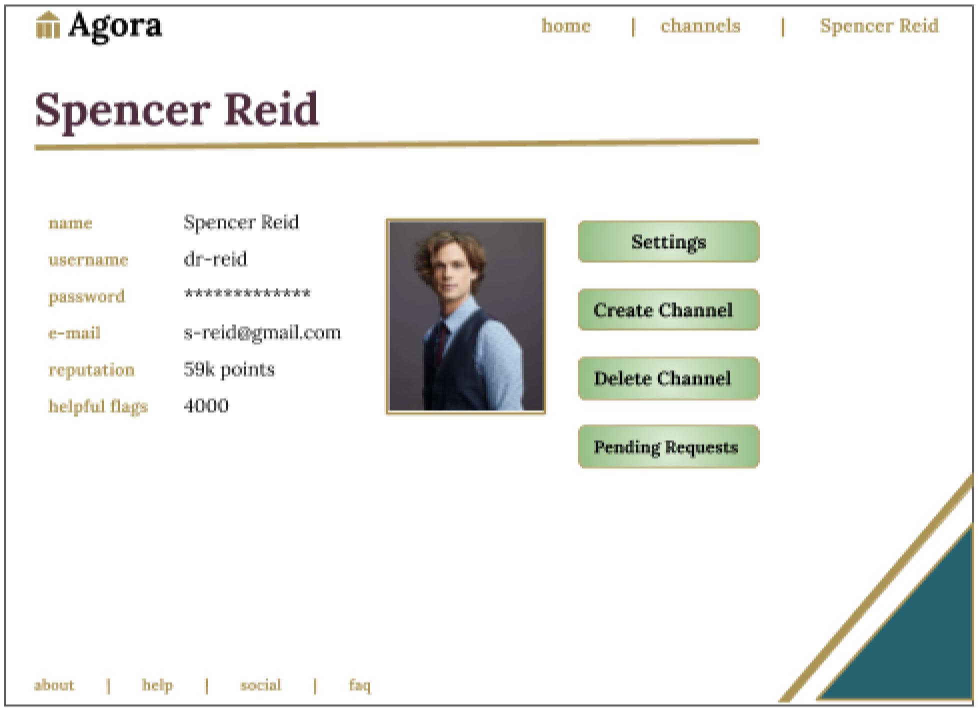
```
insert into user values(@username, @password, @email, @name, @picture_path) where @password =
@confirmpass;
insert into reputation(ID) values(select ID from user where username=@username);
```

(alternative way to check existence and create account)

```
insert into user(username, password, email, name, picture)
select * from (select @username, @password, @email, @name, @picture_path) as tmp
where not exists(
    select username from user where username = @username and @password = @confirmpass
) limit 1;

insert into reputation(ID) values(select ID from user where username=@username);
```

4.4| Profile Page

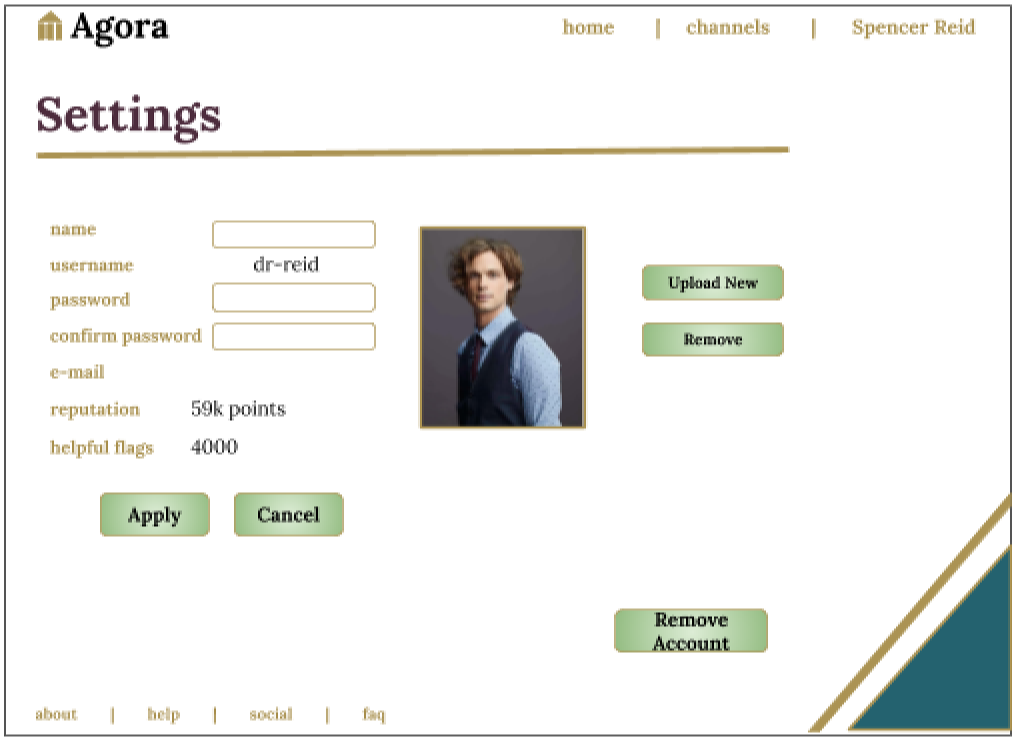


name	Spencer Reid
username	dr-reid
password	**************
e-mail	s-reid@gmail.com
reputation	59k points
helpful flags	4000

Inputs: @username
Process: The profile page represents the user information and his/her reputation as it is described above. If the user has the required amount of helpful flags, the buttons to create channel and to delete channel will be activated, otherwise they will remain deactivated. The settings button will allow user to change his or her profile information via settings screen.
SQL statements:
(Retrieve information)

```
select * from user where username = @username;
```

4.5| Settings Page



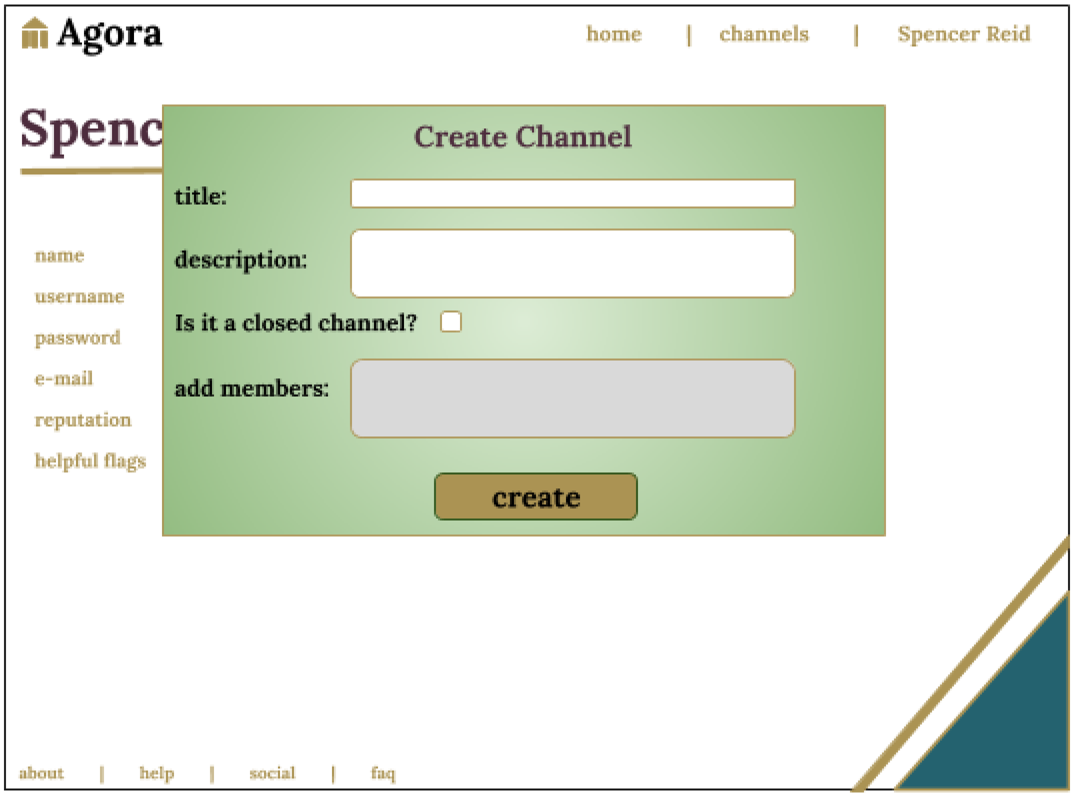
Inputs: @username, @password, @confirmpass, @email, @name, @picture_path, @defaultpic_path
Process: The settings page will be accessible through the profile. User can change his/her full name, e-mail address, password and can upload a new profile picture. If the user removes the existing profile picture and does not upload any profile picture, logo of Agora will be the profile picture. Username can not be changed once the account is created.
SQL statements:
(delete account)

```
delete from user where username=@username;
delete from subscribes where id=(select id from user where username=@username);
```

like it is deleted from subscribes its record from all the channels, posts and its relations will be deleted like this too.
(apply settings)

```
update user
set name = case when @name is not null then @name else name end,
email = case when @email is not null then @email else email end,
password = case when @password is not null then @password else name end,
picture = case when @picture_path is not @defaultpic_path then @picture_path else @defaultpic_path
end
where username = @username, @password = @confirmpass;
```

4.6| Create Channel Pop-Up



Inputs: @title, @description, @member, @username

Process: Create channel pop-up allows user to create channels like the one in the channel page. User needs to enter title and description of the channel. If the user wants the channel to private to specific members, then he/she can mark the checkbox titled “Is it a closed channel?”. Add members option will be activated if and only if this checkbox is marked. User can add members with their usernames seperated with a semicolon. If there is no one with that username then it will simply ignore that specific username and admins are automatically will be added to every channel.

SQL statements:

(check if channel already exists)

```
select id from channel where name=@title;
```

(create channel)

```
insert into channel(name,description) values(@title, @description);
select LAST_INSERT_ID() as varlast;
insert into owns_channel values((select id from user where username=@username),varlast);
```

(alternative way to check existence and create channel)

```
insert into channel(name,description)
select * from (select @title, @description) as tmp
where not exists(
    select name from channel where name = @title) limit 1;
insert into owns_channel values((select id from user where username=@username),varlast);
```

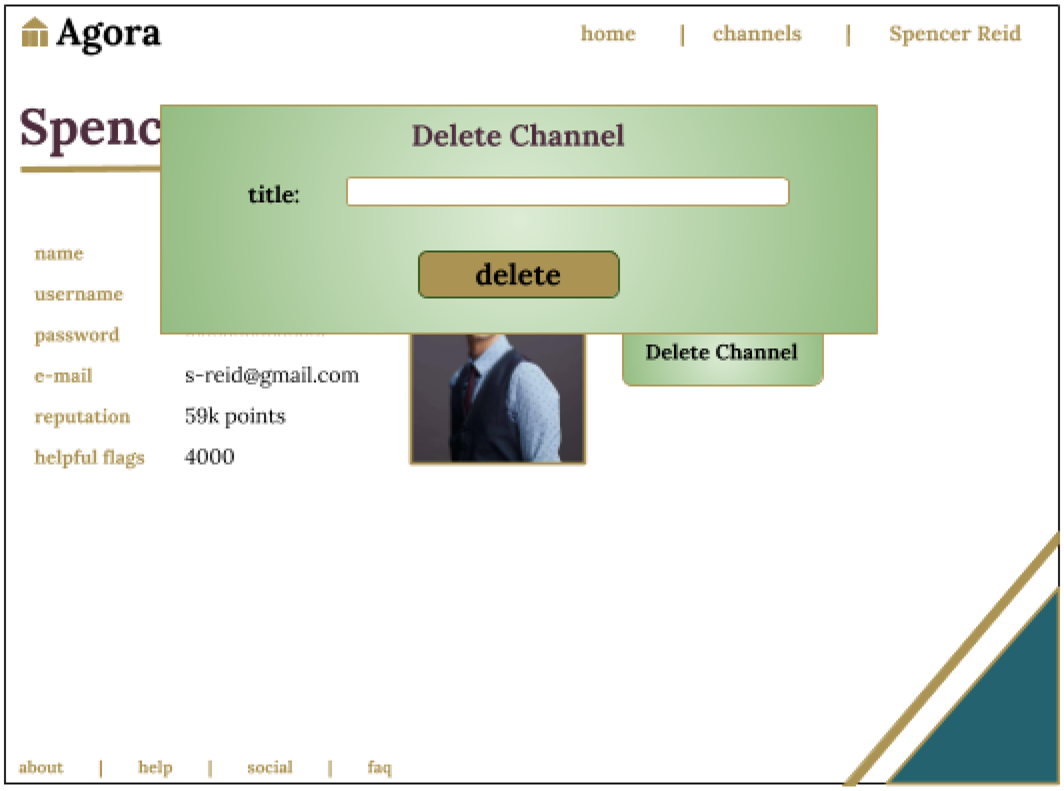
(create private channel)

```
insert into privatechannel values(varlast);
```

(add a member):

```
select id from user where username = @member;
insert into subscribes values(id,tmp);
```

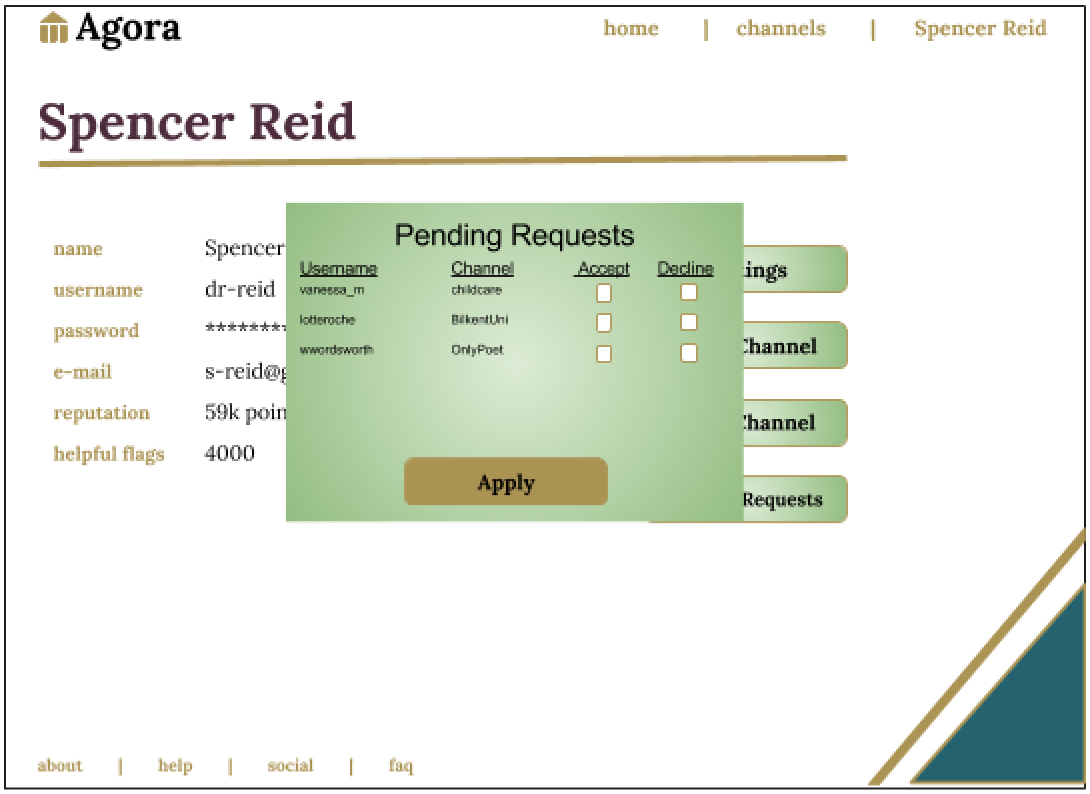
4.7| Delete Channel Pop-Up



Inputs: @title
Process: delete channel pop-up allows user to create channels like the one in the channel page. User needs to enter the title of the channel. User can delete only the channels created by him. If there is no one with that username then it will simply ignore that specific username and admins are automatically will be added to every channel.
SQL statements:
(delete channel and subscriptions)

```
delete from channel where name=@title
```

4.8| Pending Requests Pop-Up



Spencer Reid

name	Spencer
username	dr-reid
password	********
e-mail	s-reid@g
reputation	59k poin
helpful flags	4000

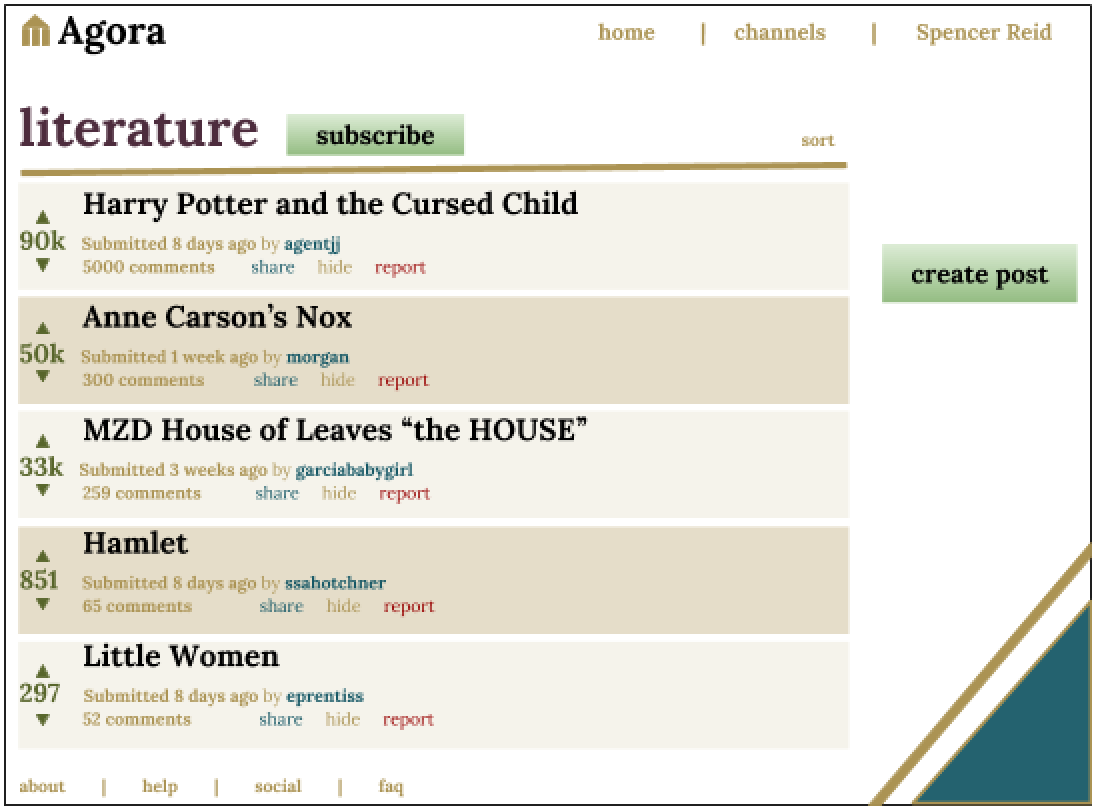
Pending Requests

Username	Channel	Accept	Decline
vanessa_m	childcare	☐	☐
lotteroche	BilkentUni	☐	☐
wwordsworth	OnlyPoet	☐	☐

Inputs: @username, @title
Process: It displays the requests to the private channels allowing user to add members to his/her channels like the one in the channel page. User is expected to accept or decline the requests.
SQL statements:
-Retrieval of the user ID and channel ID will be similar to the ones above (create account, create channel) and their subscription will be added to the system in a similar fashion in add a member in create channel.
(add a member):

```
select id from user where username = @member;
select id as tmp from channel where name=@title;
insert into subscribes values(id,tmp);
```

4.9| Channel Page



Agora home | channels | Spencer Reid

literature subscribe sort

create post

90k Harry Potter and the Cursed Child
Submitted 8 days ago by agentjj
5000 comments share hide report

50k Anne Carson's Nox
Submitted 1 week ago by morgan
300 comments share hide report

33k MZD House of Leaves "the HOUSE"
Submitted 3 weeks ago by garciababygirl
259 comments share hide report

851 Hamlet
Submitted 8 days ago by ssahotchner
65 comments share hide report

297 Little Women
Submitted 8 days ago by eprentiss
52 comments share hide report

about | help | social | faq

Inputs: @member, @posttitle, @channeltitle
Process: A non-registered user and a registered user is greeted with this page when a specified channel is chosen. In the above example, the literature channel is chosen. Only a registered user can subscribe to a channel, which means that the user is notified when a new post is added to the channel. A user can also upvote or downvote a post.
SQL statements:
(subscribe):

```
select id from user where username = @member;
select id as tmp from channel where name=@posttitle;
insert into subscribes values(id,tmp);
```

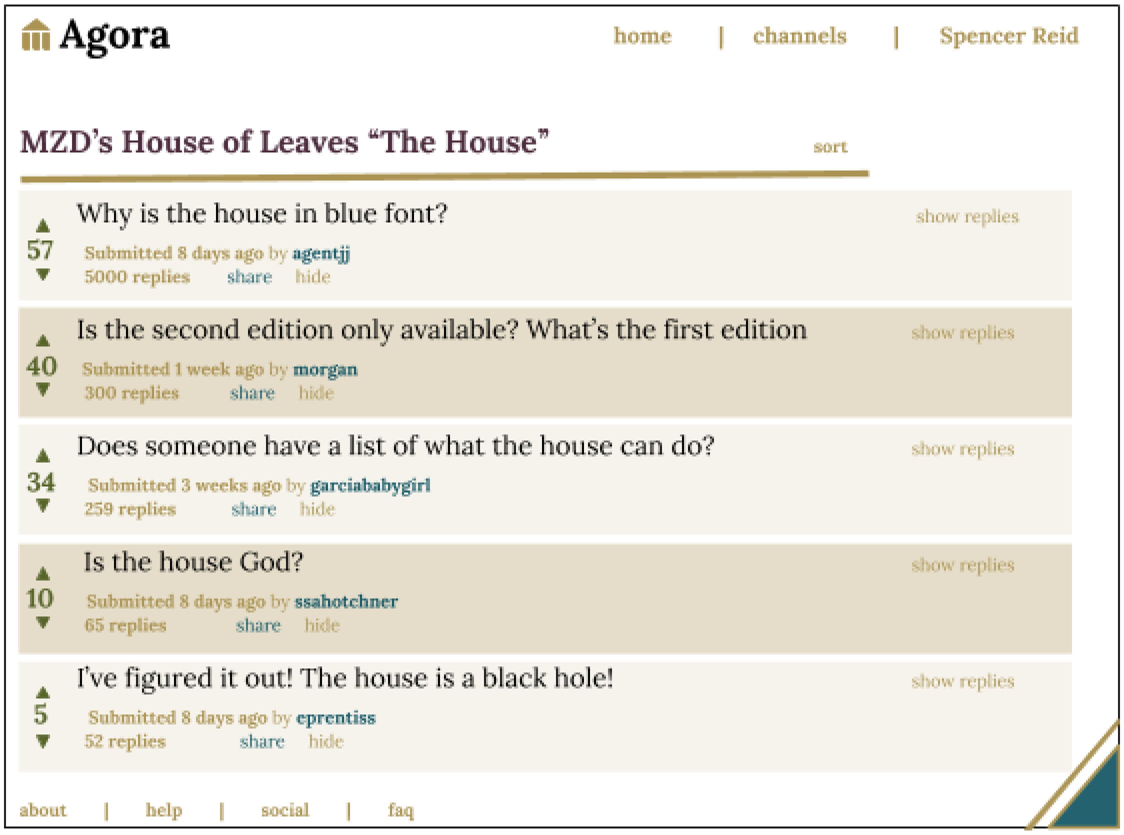
(condition to allow user to delete post)

```
select id from user t1, posts_post natural join channel natural join has_post t2 where
t1.id=t2.userID and t2.name=@channeltitle t2.title=@posttitle
```

-if the user owns the post and channel title matches with the post record then user can delete it
(delete post)

```
Select id from post where name=@title
delete from where name=@title
Delete from post
```

4.10| Unregistered User Channel Page


Agora

home | channels | Spencer Reid

MZD's House of Leaves "The House"

sort

▲ 57 ▼ Why is the house in blue font?
Submitted 8 days ago by agentjj
5000 replies share hide
show replies

▲ 40 ▼ Is the second edition only available? What's the first edition
Submitted 1 week ago by morgan
300 replies share hide
show replies

▲ 34 ▼ Does someone have a list of what the house can do?
Submitted 3 weeks ago by garciababygirl
259 replies share hide
show replies

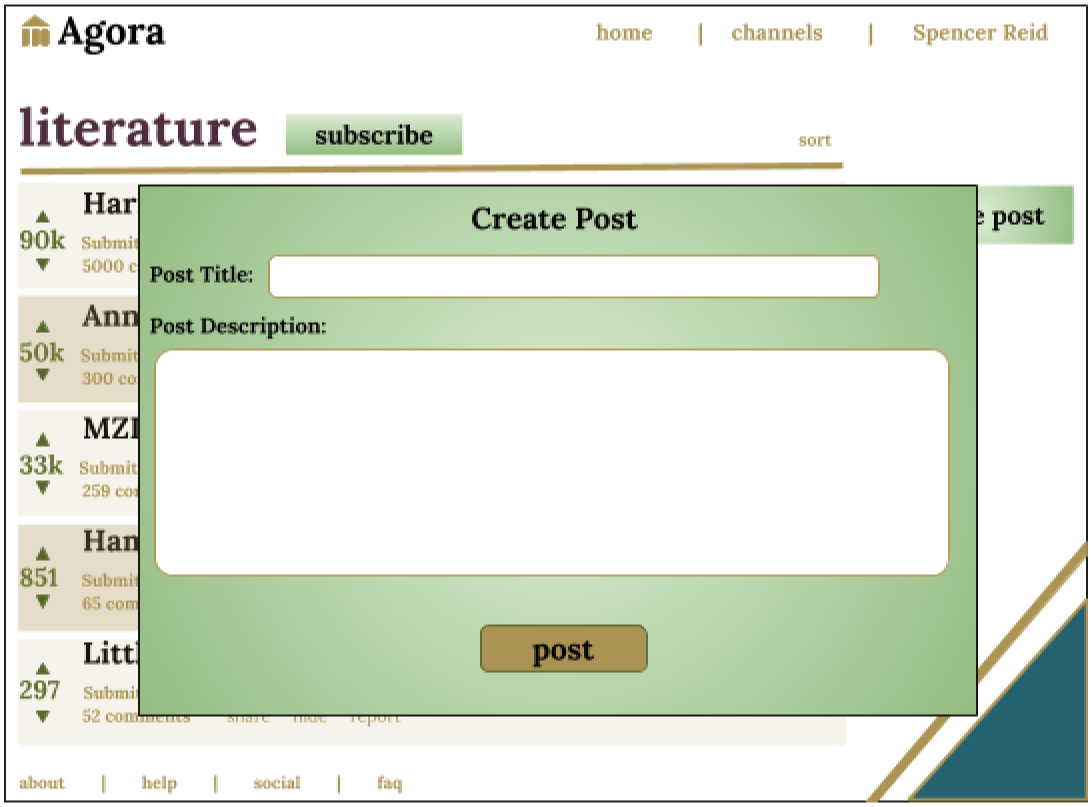
▲ 10 ▼ Is the house God?
Submitted 8 days ago by ssahotchner
65 replies share hide
show replies

▲ 5 ▼ I've figured it out! The house is a black hole!
Submitted 8 days ago by eprentiss
52 replies share hide
show replies

about | help | social | faq

Inputs: None
Process: A non-registered user and a registered user is greeted with this page when a specified post is chosen. In the above example, the "MZD's House of Leaves The House" post is chosen.
SQL statements: None

4.11| Create Post



Inputs: @title, @body

Process: Create post pop-up on the specified channel page allows user to create posts. User needs to enter title and description of the post.

SQL statements:

(check if post already exists)

```
select id from post where name=@title;
```

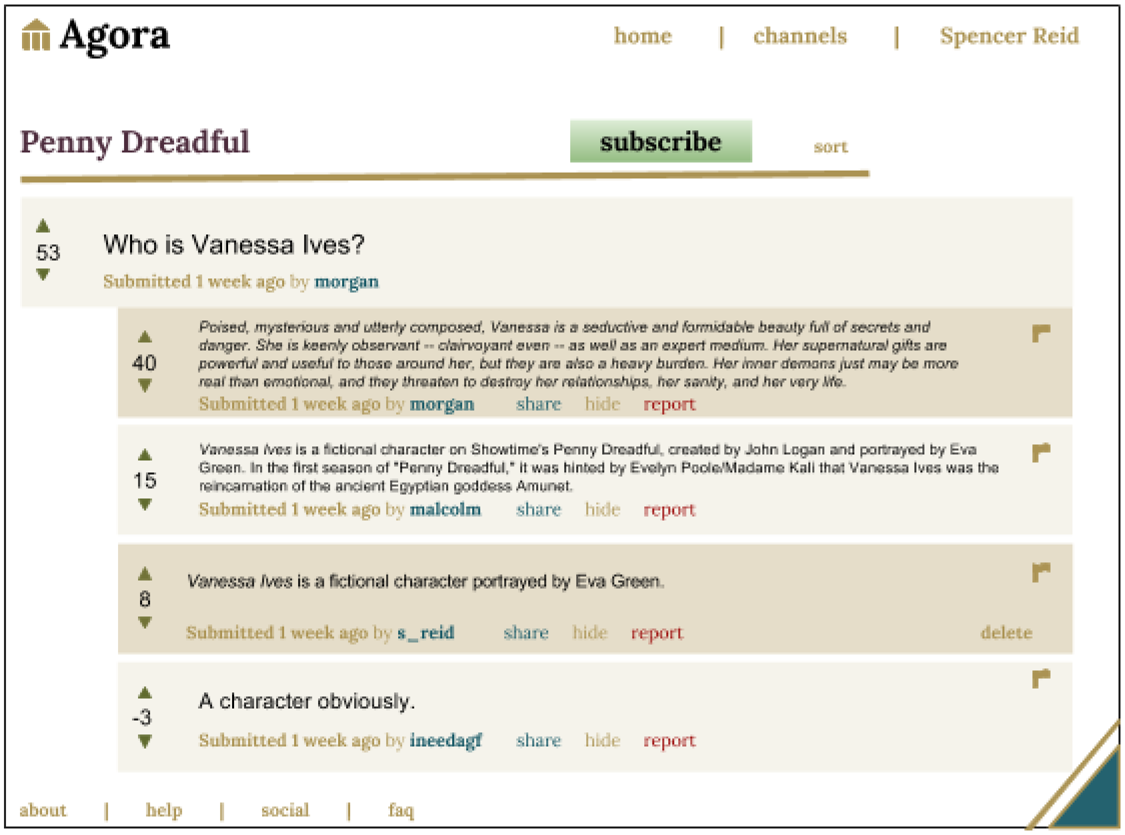
(create post)

```
insert into post(name,body) values(@title, @body);
select LAST_INSERT_ID() as varlast;
insert into owns_post values((select id from user where username=@username),varlast);
```

(alternative way to check existence and create post)

```
insert into post(name,body)
select * from (select @title, @body) as tmp
where not exists(
    select name from post where name = @title) limit 1;
insert into owns_post values((select id from user where username=@username),varlast);
```

4.12| Show Replies Page



Agora

home | channels | Spencer Reid

Penny Dreadful

subscribe

sort

53

Who is Vanessa Ives?

Submitted 1 week ago by morgan

40

Poised, mysterious and utterly composed, Vanessa is a seductive and formidable beauty full of secrets and danger. She is keenly observant -- clairvoyant even -- as well as an expert medium. Her supernatural gifts are powerful and useful to those around her, but they are also a heavy burden. Her inner demons just may be more real than emotional, and they threaten to destroy her relationships, her sanity, and her very life.

Submitted 1 week ago by morgan share hide report

15

Vanessa Ives is a fictional character on Showtime's Penny Dreadful, created by John Logan and portrayed by Eva Green. In the first season of "Penny Dreadful," it was hinted by Evelyn Poole/Madame Kali that Vanessa Ives was the reincarnation of the ancient Egyptian goddess Amunet.

Submitted 1 week ago by malcolm share hide report

8

Vanessa Ives is a fictional character portrayed by Eva Green.

Submitted 1 week ago by s_reid share hide report delete

-3

A character obviously.

Submitted 1 week ago by ineedagf share hide report

about | help | social | faq

Inputs: @posttitle, @channeltitle

Process: When the user clicks on the title of a post, he will see this screen including the title and the related comments.

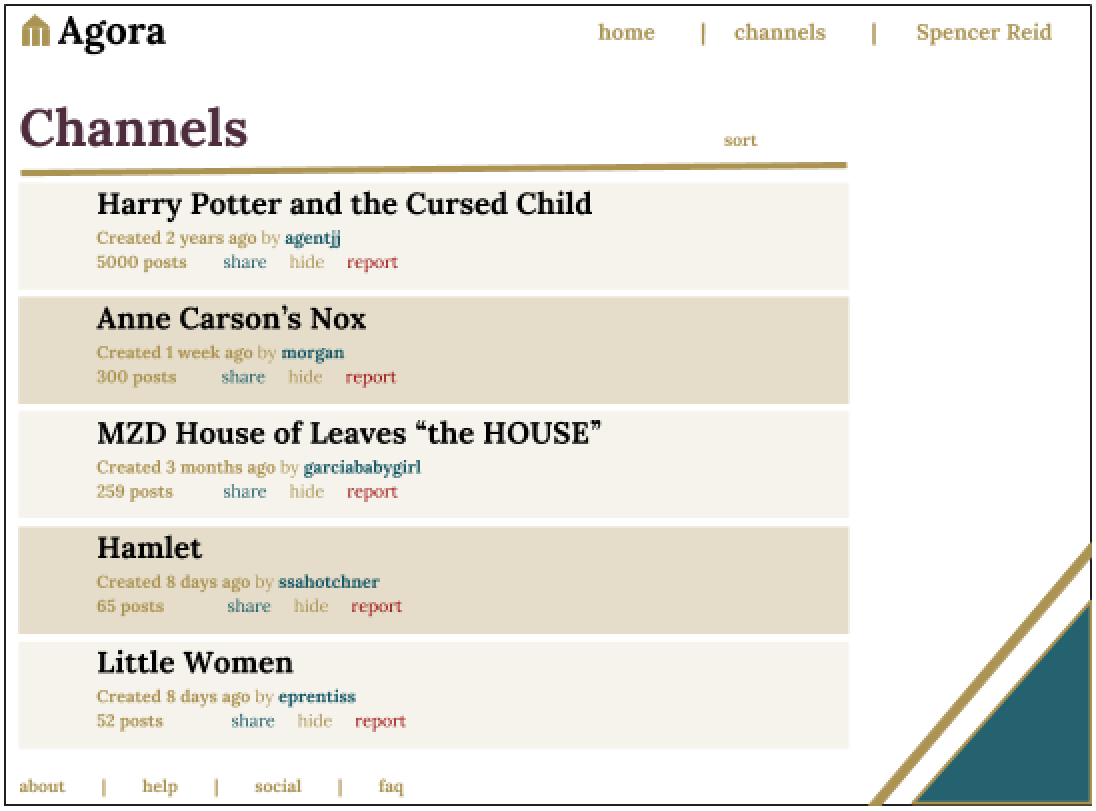
SQL statements:

-Deletion/Creation of a comment is similar to the Creation/Deletion of a post. They have similar statements differing only with their table and instance names.

(retrieve comments)

```
Select * from has_comment natural join comment natural join post natural join has_post where
title=@posttitle and name=@channeltitle
```

4.13| Channels Page



Channels

Harry Potter and the Cursed Child
Created 2 years ago by agentjj
5000 posts share hide report

Anne Carson's Nox
Created 1 week ago by morgan
300 posts share hide report

MZD House of Leaves "the HOUSE"
Created 3 months ago by garciababygirl
259 posts share hide report

Hamlet
Created 8 days ago by ssahotchner
65 posts share hide report

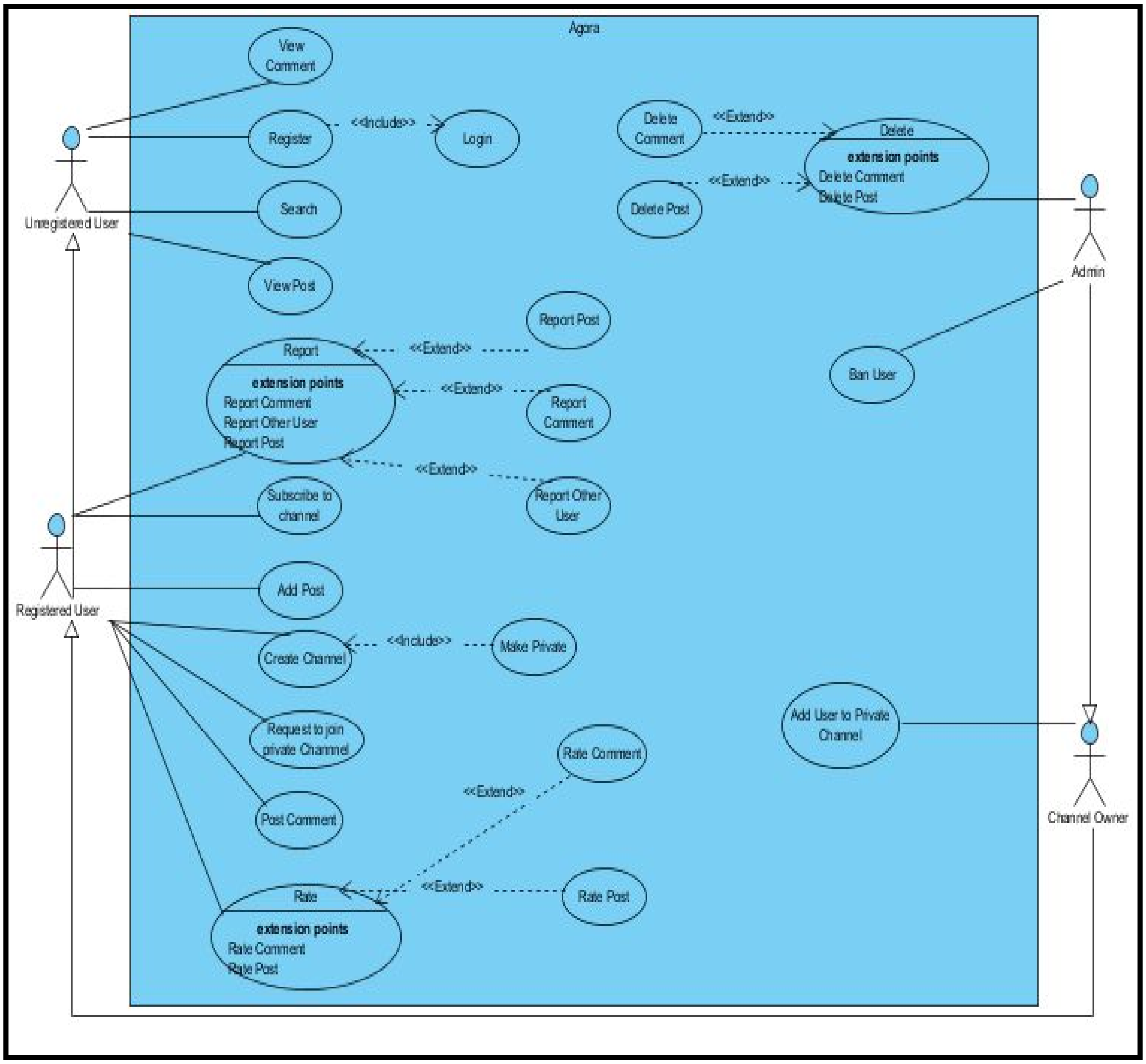
Little Women
Created 8 days ago by eprentiss
52 posts share hide report

Inputs: @username, @channeltitle
Process: When the user clicks on channels button, he will see this screen including the title of the subscribed channels.
SQL statements:
-as described in views
(retrieve subscriptions)

```
Select * from subscribes natural user natural join channel where username=@username
```

5| Functional Components

5.1| Use Case/Scenarios



The Following are the use Case of Agora and the scenarios as provided below:

Unregistered Users:

- Unregistered Users should be able to view the post and comments.
- Unregistered Users should be able to register themselves or login.
- Unregistered Users should be able to search the website for the appropriate forum/post they please.

Registered User

Registered Users are defined as users who have registered/logged in into the system. These Registered have are the same as Guest but they have extended rights as presented below:

- Registered users have a right to subscribe to a channel.
- Registered users should be able to subscribe to a channel.
- Registered users should be able to create a post.
- Registered users should be able to comment in a post.
- Registered users should be able to rate a comment as helpful or not helpful.
- Registered users should be able to rate a post as helpful or not helpful.
- Registered users should be able to report a Post or comment they feel inappropriate.
- Registered users have a right to report another user.
- Registered users have a right to send a request to join a closed forum.
- Registered users should be able to Create a channel.

Post Owner

In order to deal with private channels we created Post Owners users which have the same rights as registered Users but are extended upon their ability to add users to a forum

- Post Owners should be able to add other users to their private forum.

Admin

Admins are superior users as compared to registered users. They can do anything including:

- Admins should be able to remove a Comment that was reported.
- Admins should be able to remove a post with substantial reports.
- Admins should be able to ban a user with substantial reports.
- Admins should be able to Unban a user if, when they feel appropriate.

5.2| Algorithms

Reputation Algorithm

Most of the discussion platforms are suffering from piled up redundant posts/channels. Some users may try to spoil the platform by creating those redundant entries. Agora has a way to prevent that problem.

Agora have built-in channels including generalized categories such as music, literature etc. Every user account is linked to a reputation entry once it is created. The values of this reputation entry are initialized to be zero by default. When someone marks any user's post with a helpful flag, his/her reputation points are going to be increased. Once one reaches the sufficient amount of reputation points, then he/she may be able to create new channels.

Any reported entry may decrease the user's reputation according to the report type. Even she/he can be deleted/banned from the system by admins according to the report type he/she gets.

5.3| Data Structures

The relation schemas we have created uses Numeric Types, Boolean Types, String Types, and Date and Time Types.

- Numeric types:INT
- Boolean Types: BOOLEAN
- String types: CHAR, VARCHAR
- Date and Time: TIMESTAMP

6 | Advanced Database Components

6.1 | Views

6.1.1 | View user history:

The following will get the list of all the activities that the user has done, since they first started using the sytem.

```
CREATE VIEW show_user_history AS
SELECT object_postedID, time_posted
FROM
(SELECT
channelID AS object_postedID,
time_posted
       FROM Owns_Channel NATURAL JOIN  (
              SELECT
ID as channelID,
since AS time_posted
              FROM Channel)
WHERE userID = @userID) NATURAL JOIN

(SELECT
ID  AS object_postedID,
time_posted
       FROM Post NATURAL JOIN  (
              SELECT postID AS ID,
              FROM Posts_Post
              WHERE userID = @userID
) NATURAL JOIN (

SELECT
ID  AS object_postedID,
time_posted
       FROM Comment NATURAL JOIN  (
              SELECT commentID AS ID,
              FROM Posts_Comment
              WHERE userID = @userID)
)
GROUP BY time_posted DESC;
```

Note: @userID is the unique ID for that specific user

6.1.2| View all channel that the user is subscribed to:

This view as an operation to show all the channels that a single user is subscribed to

```
CREATE VIEW show_subscribed_channels AS
       SELECT name AS Channels
       FROM Channel NATURAL JOIN (
SELECT ChannelID
FROM Subscribes
Where userID = @userID
);
```

Note: @userID is the unique ID for that specific user

6.1.3| View pending request to join a private channel

The following view will be used as the notification at the top left corner to display all the users who wishes to join the private channel owned by that specific user

```
CREATE VIEW show_pending_request AS
SELECT userID
FROM  Request_Follows NATURAL JOIN  (
SELECT channelID
              FROM Owns_Channel
              WHERE Owns_Channel.userID = @userID)
WHERE IsApproved == 'False';
```

6.2| Reports

6.2.1| View user's ranking according to their reputation (the best are at top):

Get list of all the user where the user which has the worst reputation, meaning the highest number of reports will be displayed at the top of the list

```
CREATE VIEW show_users_reputation AS
SELECT userID, total reputation AS reputation
FROM Reputation
GROUP BY reputation DESC;
```

6.2.2| View user's ranking according to number of reports they received (worst at top):

Get the list of all the users who have the best reputations

```
CREATE VIEW show_users_reported AS
SELECT userID, number_of_reports AS reports
FROM Reputation
GROUP BY reports DESC;
```

6.2.3| View user's ranking according to number of useful flags they received (best at top):

```
CREATE VIEW show_users_usefulness AS
SELECT userID, useful_flags
FROM Reputation
GROUP BY useful_flags DESC;
```

6.3| Triggers

- When a user upvotes or downvotes a post (or a comment), the rating of the post (or a comment) will be updated. Additionally the ‘up_down_votes’ corresponding to the user that posted it will be updated as well. This ‘up_down_vote’ attribute will affect ‘total_reputation’ attribute of the user as it is calculated on the bases of it.
- When a new post is added by the user, the ‘Post’ entity set will be updated accordingly. In this entity set the user is required to provide a ‘title’ and a ‘body‘
- When a user adds a comment , the ‘Comment’ entity set will be updated accordingly. In this entity set the user cannot leave the entity as empty but instead are required to provide a body.
- When User adds a new Channel, the ‘Channel’ list and the ‘Owns_channel’ get updated.
- Before a user can add a post/comment in a channel there is a trigger event to check whether the post is in a private or public channel. If private there is a trigger event to check whether the user is part of that channel or not.
- Before a user can be added to the banned table , a trigger event would be run inorder to ensure the user who is banning(i.e adding the other use to the banning list) is indeed the admin of the system.

6.4| Stored Procedures

We plan to use stored procedures to initialize some attributes and entities. We plan to use stored procedures when creating a user, a channel and a post or a comment. For each of those we need to write those queries over and over again, instead of having to write that queries each time we are going to save those as a stored procedure and then just call the stored procedure to execute the SQL code that we have saved as part of the stored procedure.

Deletion of a user or a channel requires that deletion queries to run over and over again from multiple tables. Therefore we are going to use a stored procedure in case of deletion of a user, a channel etc to reverse their effect on database.

6.5| Constraints

- The system does not allow a comment or post once it has been posted.
- If a comment is a reply, then the comment can only be posted if its reply time is less than the reply time of master comment.
- If an admin bans a user, the system only allows the user to come back as a non-registered user.
- The system will not allow users to post a forum or comment, report, rate a forum or comment, and create a profile if they have not logged into their account.
- The system allows the user to rate the same post only once.
- The system allows the use of only 40 downvotes/upvotes (forum, comment or rating point in a day.
- The system permits the user to post only one profile picture.

7 | Implementation Plan

We plan to use MySQL for our database management system, and PHP for the development technology. We plan to use Javascript, Java, CSS, HTML to implement the user interface. We also plan to use Java to provide an interface between front-end user-interface and the back-end database.There are no specific hardware systems needed for our implementation. Any average computer or laptop with the environment to run our specific software systems can be used.

8| References

1. “What is database and why do we need them?” http://www.softwaretestingclass.com/what-is-database-and-why-do-we-need-them/