MUNSSN

Software Architecture Document

Version <1.0>

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Version** | **Description** | **Author** |
| 24/Feb/17 | 1 | First revision of architecture document | Floris Bouman, Josh Davis, Bradley Gavan, Matthew Randall, and Curtis White |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Table of Contents

1. Introduction 2

1.1 Purpose 2

1.2 Scope 2

1.3 Overview 2

2. Architectural Representation 2

3. Architectural Goals and Constraints 2

4. Use-Case View 2

4.1 Use-Case Realizations 2

5. Implementation View 2

6. Size and Performance 2

7. Quality 2

Software Architecture Document

# Introduction

The purpose of this document is to graphically present the structure of each of the modules in the system software of the MUNSSN. A module is a unit of the software which can be considered independently encapsulated from other modules. A module may depend on aspects of other modules, but can be modified without affecting the functionality of the other modules. Each module is the responsibility of a different team member. The modules and the member responsible for it will be outlined below.

## Purpose

This document provides a comprehensive architectural overview of the system, using a number of different architectural views to depict different aspects of the system. It is intended to capture and convey the significant architectural decisions which have been made on the system. The modules are primarily explained through sequence and class diagrams.

## Scope

This document provides a visual overview of the system and how it is structured. These visual representations help team members, as well as those interested in the project, interpret the project. Coding design will lead largely from these visuals.

## Overview

Below one will find the architectural representations of the software, primarily the use case diagram of the system, as well as sequence diagrams representing the flow of each module.

# Architectural Representation

Each module is represented by a sequence diagram which is to complement the use case diagrams contained in the Software Requirements Specification. The entire system is represented by a use case diagram with each module given a connection to the others depending on its functionality.

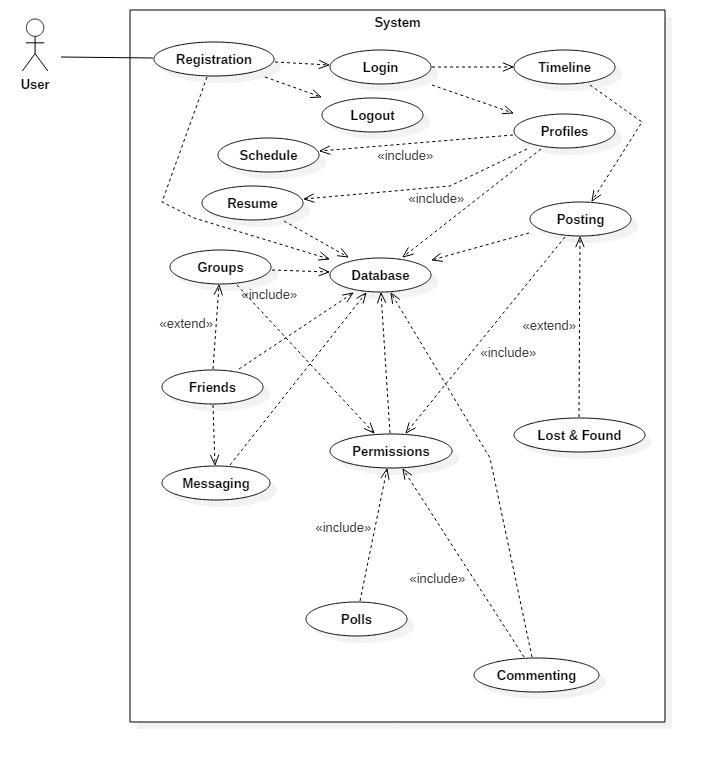
# Architectural Goals and Constraints

The modules must ultimately provide users with an experience that ensures privacy, stability, and reusability. Users must only be able to access their own accounts through normal operation of the system. The user's data must be protected from loss and must be presented to the user reliably. The system's modules must be reusable so that expansions to the system can proceed efficiently without breaking the system.

# Use-Case View

The system is presented below as an overarching use case whereby the user logs in and all the modules can be seen by their connections to other modules.

## Use-Case Realizations



# Implementation View

The following section shows the system as modules, and the flow of each module is visualized as a sequence diagram. Within each section, a member is assigned responsibility of the module, and pseudocode is provided to elaborate on the module's flow which is visualized in the sequence diagram.

1. **Registration – Curtis White**

Signatures:

* register(String name, String username, String password, String e-mail, String courses) - User inputs and sets each String value that makes up the user’s information. If all values are valid, user is added to database. Username and password will be used to log in to each session.
* isRegistered(String username, array users[]) - Checks username against database of users and returns a boolean value which specifies whether the user has been registered or not. This will be used to determine whether user exists to display appropriate error messages.

The registration process is started by having the user click “Sign Up” on the MUNSSN start page. A form appears which is to be filled out. The inputted values are posted to the register() method where they are checked for validity (username is passed to isRegistered() and a boolean is returned to register()). If all values are valid, the user is added to the database.

void register(String name, String username, String password, String e-mail, String courses) {

if (isRegistered(username) or isRegistered(e-mail) is true) {

error message, username already exists

} else {

if (name, username, e-mail, password are valid format) {

add user to database

} else {

error, correct input and try again

}

}

boolean isRegistered(String username, array users[]) {

if (username exists in users[]) {

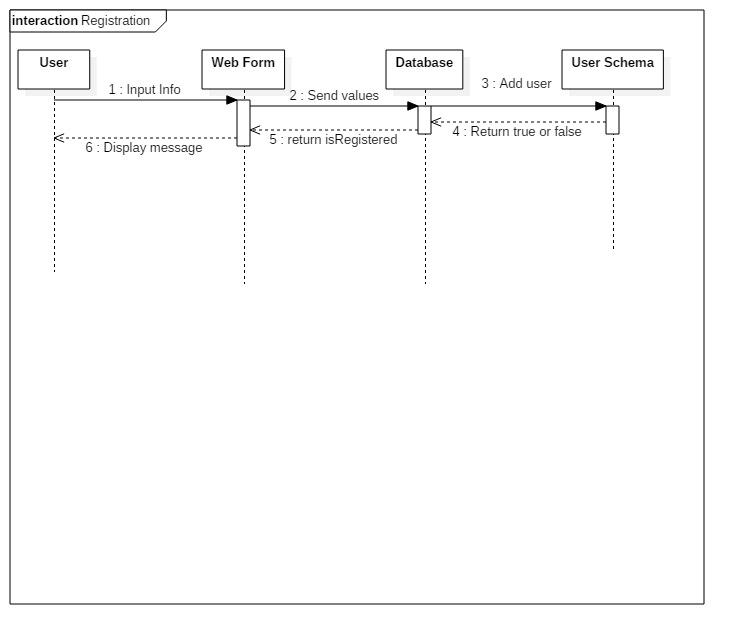
return true

} else {

return false

}

}



2. **Login/Logout - Curtis White**

Signatures:

* login(String username, String password) - User inputs username and password. If valid, user is entered into a session and gains access to their account content.
* logout() - Session is closed. User must log in to regain access to account.

The login process is started by having the user click “Log In” on the MUNSSN start page after entering their username and password. If valid, user is taken to their account’s start page.

The user clicks “Log Out” on their account’s start page to log out. This closes the session, takes them back to the MUNSSN login page and the user must login again to regain access to their account.

void login(String username, String password) {

if (username and password are valid) {

load user’s database account

take user to account start page

} else if (username or password are invalid) {

error, invalid user credentials

          } else if (username not found in database) {

error, user does not exist

}

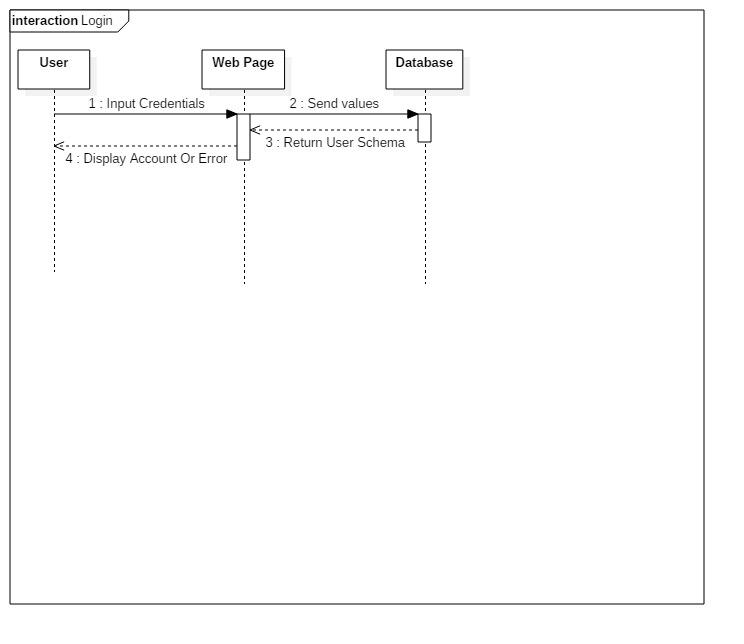
}

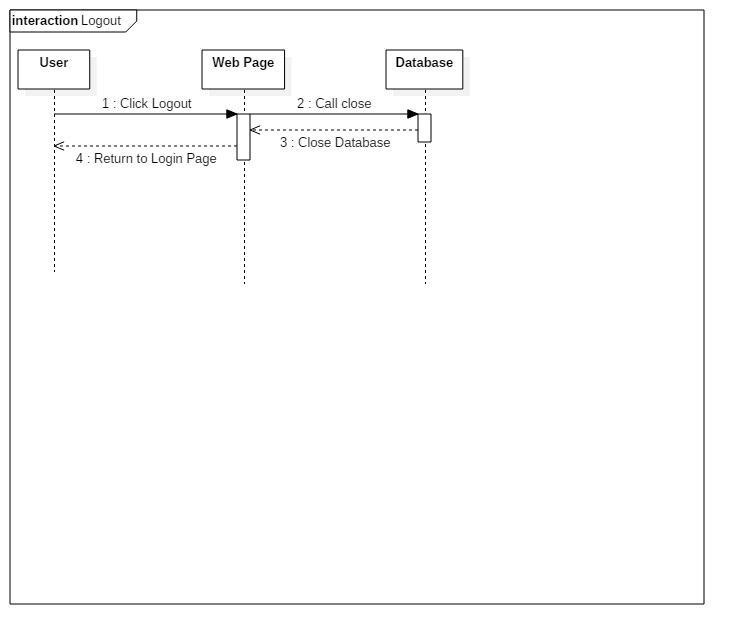
void logout() {

close database session

take user to login page

}





3. **Timeline - Curtis White**

Signatures:

* loadPosts(posts[]) - Upon viewing the timeline, posts stored in the database are pulled and displayed on the page.
* makePost(String post) - Typed String is inputted into a textarea, and is then stored in the database when the User clicks “Submit.”
* editPost(String post) - User can edit a submitted post by clicking “Edit” and its content will be overwritten in the database. loadPosts() is called again to adjust the posts displayed on the timeline.
* deletePost(posts[id]) - User can click “Delete” on their post and it will be removed from the database. loadPosts() is called again to adjust the posts displayed on the timeline.

Upon logging into the MUNSSN start page, the timeline is loaded and set number (default 10, adjusted by user settings) of posts fill the timeline <div> structure.

To post, the user types a message in the textarea above the timeline and clicks “Submit.” The timeline is updated with the new post.

To edit post, the user clicks edit within a post <div> and the post becomes a textarea which can be edited. The user clicks “Submit” and the timeline is updated with the edited post.

To delete a post, user clicks “Delete” within post <div>, a confirmation dialog is displayed, and if user selects “Yes,” the post is removed from the database. The timeline is updated to reflect the removal of the post. If user selects “No” then the timeline returns to its most recent state.

void loadPosts(posts[]) {

if (posts[] is not empty) {

pull posts from database

display most recent 10 posts

} else {

display message, no posts

          }

}

void makePost(String post) {

if (post is not empty) {

add post to database

update timeline to show latest post

} else {

error, no post found

          }

}

void editPost(String post) {

make text <div> an editable textarea

if (submit selected) {

update post in database

} else if (Close selected) {

return textarea to standard <div>

set post to most recently stored version

}

}

void deletePost(String post) {

display confirmation message

if (user selects Yes) {

delete post from database

close message

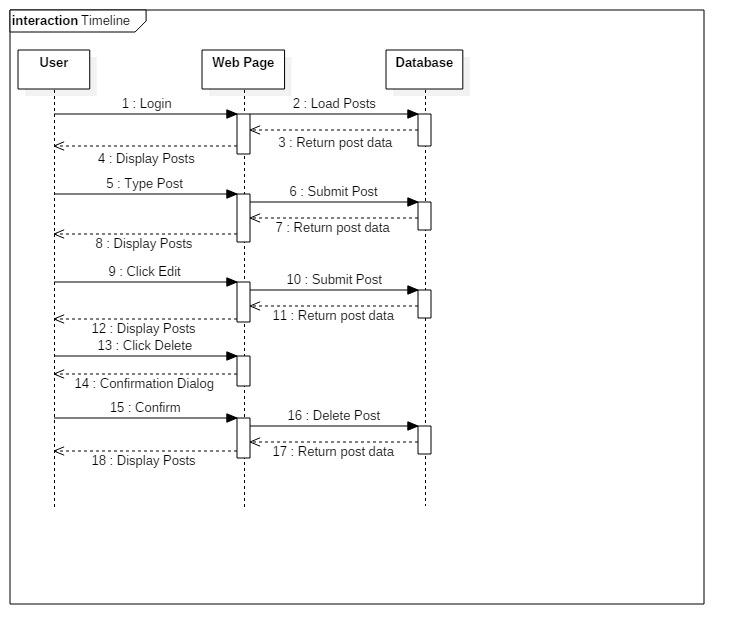
update timeline to show latest posts

} else if (User selects No) {

close message

          }

}



4. **Database - Curtis White**

Signatures:

* start() - Loads database profile in order to connect.
* connect() - Connects the database server to the localhost or proper port and routes users to the specified start page (login.js).
* close() - Closes the database connection, removing access to the database.

Upon loading the MUNSSN URL in a browser, the server is started, connects to the specified port, and routes users to the login page. Users may then login.

Once a user is finished with a session, they click logout and the database will close and they will be routed back to the login page.

void start() {

if (server started) {

// Do nothing

} else {

start server at URL

connect()

          }

}

void connect() {

if (server started) {

connect to port

route to start page

} else {

error, server not started

try starting server

          }

}

void close() {

if (server started) {

close database

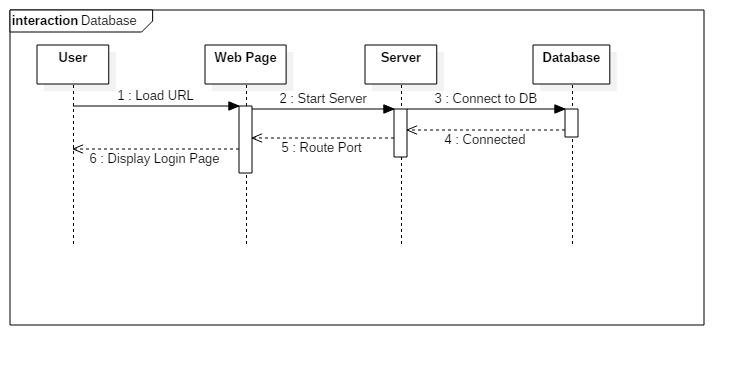
route to start page

} else {

// Do nothing

}

}



5. **Groups - Bradley**

Signatures:

* start() - Loads database profile in order to connect.
* connect() - Connects the database server to the localhost or proper port and routes users to the specified start page (login.js).
* close() - Closes the database connection, removing access to the database.

Signatures:

* createGroup(String name, String course) - New group is created
* inviteUsers(array users[]) - Notification sent to the users with a link to join the group
* addAdmin(String user) - The user (member of the group) receives admin privileges
* removeUsers(array users[]) - Users are removed from the group
* setPrivacy(String(struct) privacySetting) - The group privacy setting is changed in the database

Groups will have a similar structure to a user’s personal page. A user creates a group for a specific course offered at MUN. The creator of the group is the admin for the group. Admin’s can change descriptions on the page, remove users and content. Members of the group can invite other members. Members can post content, a group will have its own timeline (the timeline module will handle all posts). The posting module will handle posting for Groups. Who can view the group and it’s content will depend on the privacy setting of the group, which can be set by an admin.

boolean createGroup(String name, String course) {

if (Group name is not taken) {

Create an instance of a Group in the database

Set the creator as the admin

Allow the creator to invite users

return true

else {

error, name already taken.

return false

}

void inviteUser(array users[]) {

for user in users {

send notification

attach link to join group

}

}

void addAdmin(user) {

db.group.Admin add(user)

}

void removeUser(user) {

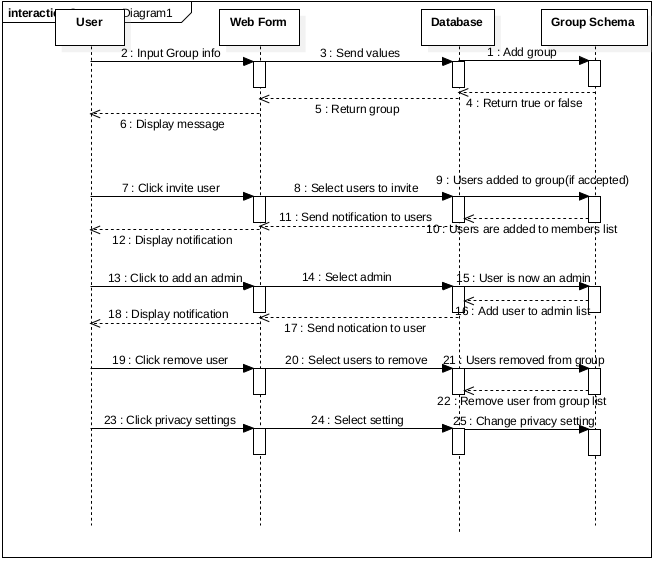
db.group.users remove(user)

}

void setPrivacy(Struct privacySetting) {

groupPrivacy = privacySetting

}



6. Lost/Found - Bradley

Signatures:

* post(title, description, location, picutre, lostOrFound) - creates a post on the pages timeline
* markFound() - changes the status of a post to “Found”
* mapLocation(location) - returns a pin on the Google Map for the given location

Lost and Found is decorated version of the group module. It is a group that will be viewable to all users, users will not be able to join/leave the group. Users will post in this group when they lose or find an object on campus. A post will be made up of a title, description of the lost/found item, the location it was found, a picture of the object and a tag indicating whether the poster found or lost the object. A post that was marked as lost can be marked as “Found” when the posters finds the object, to let other users know that they are no longer looking for it. For an object “Found” post, the user can include the location where they found it. The Google Maps API will be used for this. The poster will see a map, originally zoomed onto the Memorial Campus. They will be able to drop a pin where the found the object

void post(title, description, location, picutre, lostOrFound) {

create new post in db

post.title = title

post.description = description

post.location = location

post.picture = picture

post.lostOrFound = lostOrFound

add post to post stack for Lost and Found group

}

void markFound() {

post.lostOrFound = Found

}

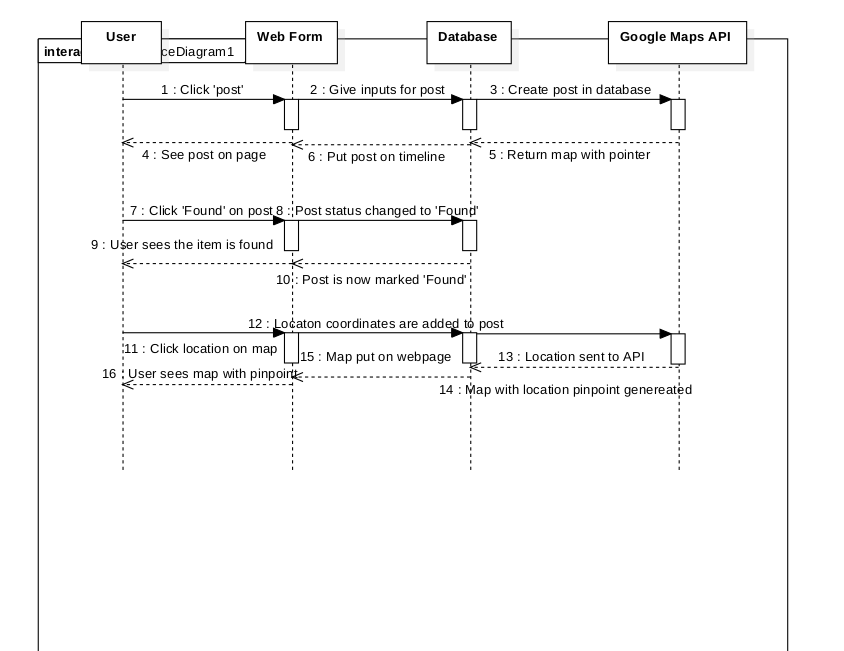
void mapLocation(location) {

marker = new google.maps.Marker( {

position: location

map: this.map

});



7. Messaging - Bradley

* sendMessage(user, message)
* deleteMessage()

Messaging is a module that allows users to send private messages between users or groups of users. A user types a message, selects a user(s) and sends it. The recipient will than receive a notification that there is a new message in their inbox.

void sendMessage(user, message) {

user.push(message) to inbox

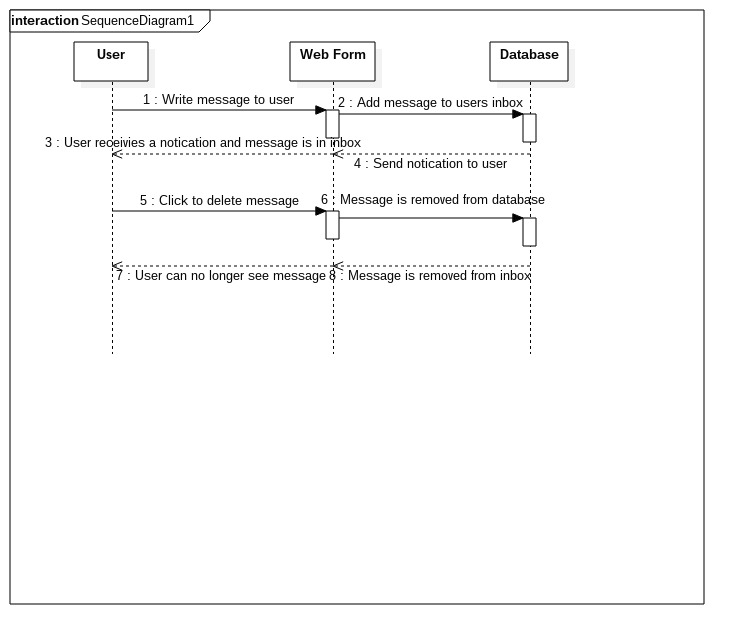
send user a notification

}

void deleteMessage(message) {

remove user.inbox.message

}



8. Friend System

**Signatures:**

readRandom() - read a random list of students from the data base

checkMajor(random list element) - check if the current student in the random list has the same major as the logged in student, return a int value to represent a score

checkMinor(random list element) - check if the current student in the random list has the same minor as the logged in student, return a int value to represent a score

checkMutual(random list element) - check if the current student in the random list has any of the same friends as the logged in student, return a int value to represent a score

checkClasses() - check if the current student in the random list has any of the same classes as the logged in student, return a int value to represent a score

createSuggestions(values from the check classes) - take in the values from the checks and arrange high scoring people at the from of a list.

displayList(suggestionList) - show the user their suggested friends

The friend system module is an algorithm for finding people on the site with similar traits as the user. suggested friends are created randomly when the user logs in and are displayed in order of highest score in the algorithm.

readRandom(){

randomList;

for(10 times){

randomList.add(db.readRandomUser());

}

}

checkMajor(random list element){

if randomListElement.getMajor() == this.getMajor()){

return 20 (max score)

}

else return 0(min score)

}

checkMinor(random list element){

if randomListElement.getMinor() == this.getMinor()){

return 20 (max score)

}

else return 0(min score)

}

checkMutual(random list element){

if randomListElement has 50 or more friends in common{

return 20 (max score)

}

if randomListElement has 25 to 49 friends in common{

return 10

}

if randomListElement has 10 to 24 friends in common{

return 5

}

else return 0(min score)

}

checkClasses(random list element){

if randomListElement has 4 or more of the same classes{

return 20 (max score)

}

if randomListElement has 2 to 3 of the same classes{

return 10

}

if randomListElement has 1 of the same classes{

return 5

}

else return 0(min score)

}

createSuggestions(values from the check classes){

add list element scores and IDs to an unordered list called suggestions

suggestions.sort() //highest value to lowest

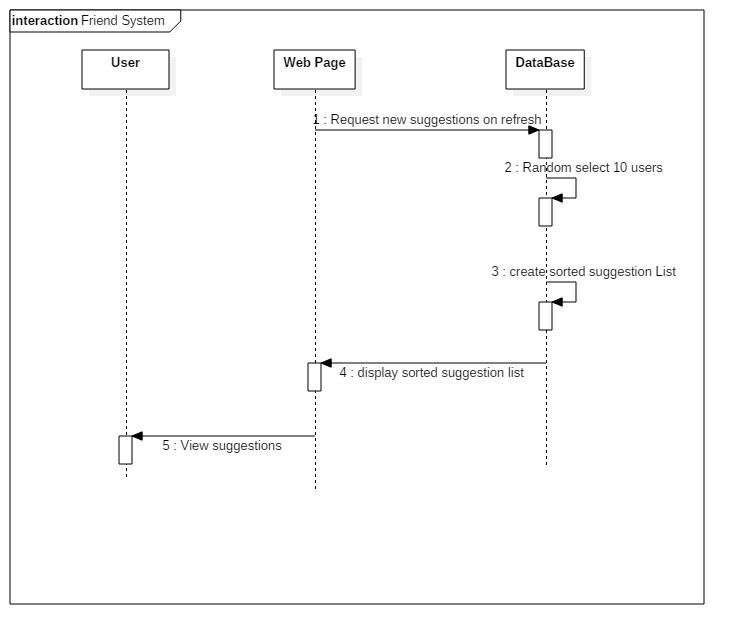
return sortedSuggestions

}

displayList(suggestionList){

timeline.display(suggestionList)

}



10. Permissions/Content Visibility – Josh

Signatures

checkPerm(String choice) – User clicks a drop down menu which give the options of sharing content with everybody, only themselves, their friends or a specific list. This will be applicable for both posts and comments.

void checkPerm(String choice){

if(choice equals me){

display content to user only

}

if(choice equals friends){

load friends list from database

push content to friends

}

if(choice equals list){

load list from database

push content to list members

}

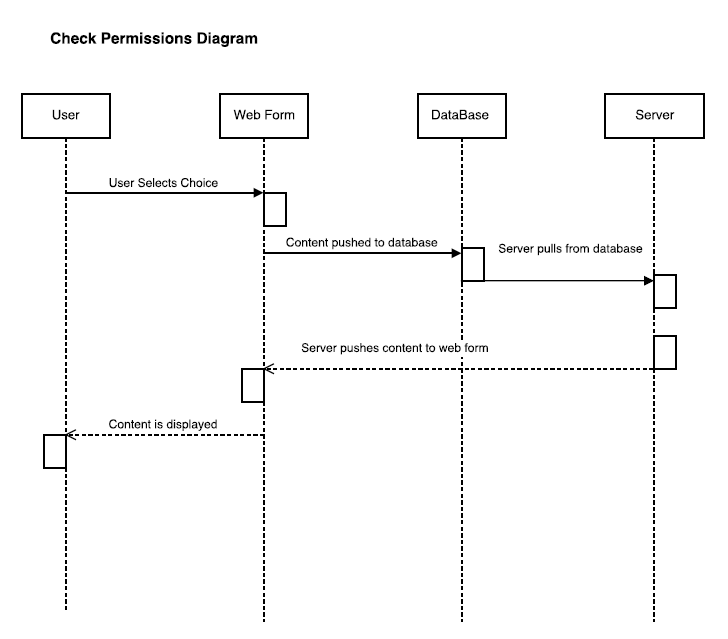
if(choice equals everyone){

display content on site

}

}

When the user creates a post he will have the option next to it to choose who they display the content too. The options will be for only the user, their friends that they have added on the site, a list they may have created on the site or to anyone that is a member of the site. This option can be changed at any time.



12. Schedule System – Josh

Signatures

void createSchedule()- uses course data from server to get days, time, course name and uses this data to fill in a schedule table.

void inputSchedule(String name, String time, String[] days) – Takes input from the user to add course information and stores it in the database. User clicks add a course and fills in necessary information.

void inputSchedule(String name, String time, String[] days){

store name, time, days

push information to database

}

void createSchedule(){

Pull information for specific user from database and store in variables

for(integer is less then days size){

if(day equals monday){

column equals 1

for(time between 7 to 22){

if(database time equals searched time){

output course name into column 1 and time slot

}

}

}

if(day equals tuesday){

column equals 2

for(time between 7 to 22){

if(database time equals searched time){

output course name into column 2 and time slot

}

}

}

if(day equals wednesday){

column equals 3

for(time between 7 to 22){

if(database time equals searched time){

output course name into column 3 and time slot

}

}

}

if(day equals thursday){

column equals 4

for(time between 7 to 22){

if(database time equals searched time){

output course name into column 4 and time slot

}

}

}

if(day equals friday){

column equals 2

for(time between 7 to 22){

if(database time equals searched time){

output course name into column 5 and time slot

}

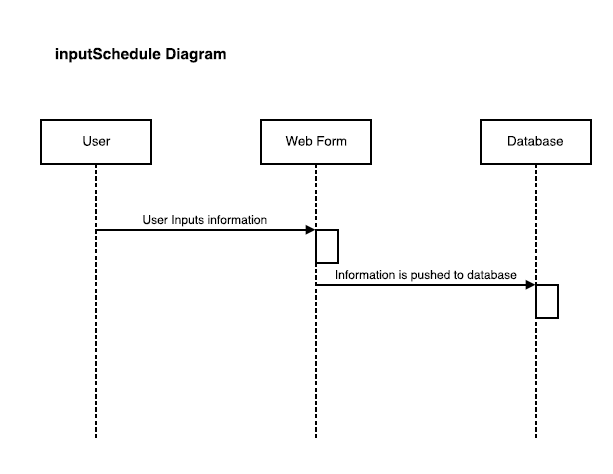
}

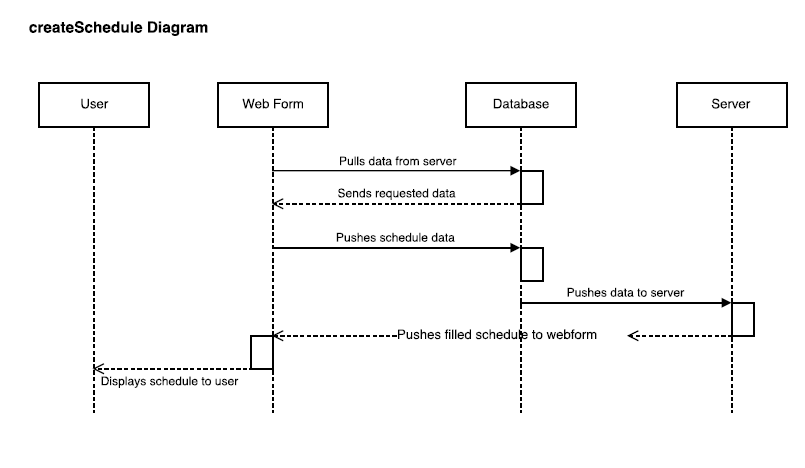
}

}

}

The user will have a button on their sidebar to add a course. This will bring them to a page with inputs for course name, days and time. Once the user submits this information they will be prompted if they want to add another course or be taken to their schedule. Once the data is entered the user's schedule will automatically be updated and can be set to have permissions for viewing similar to timeline posts.





13. Polling System – Josh

void initializePoll(String title, String question, String[] options) – takes in the polls title, question that’s being asked and the options to vote on. Send the data to the database to be stored.

void createPoll() – Uses the data stored in the database to output a poll that the user can read and vote on. Also shows how many users voted and how many each option got.

void updateVotes(radio button output) – A function to take in a user vote and update the database, which inturn updates the poll output.

void initializePoll(String title, String question, String[] options){

store inputted data

create votes tally array filled with all 0’s based off the size of options inputted

create total votes integer

push data to database

}

void createPoll(){

server pulls data from database

output title

output question

for(all question options){

create radio button

print option(i)

}

for(votes tally){

output number of votes next to option it matches up with

}

output total votes

}

void updateVotes(radio button output){

pull database data

create integer

for(options){

if (radio button output equals option){

set integer to option number

}

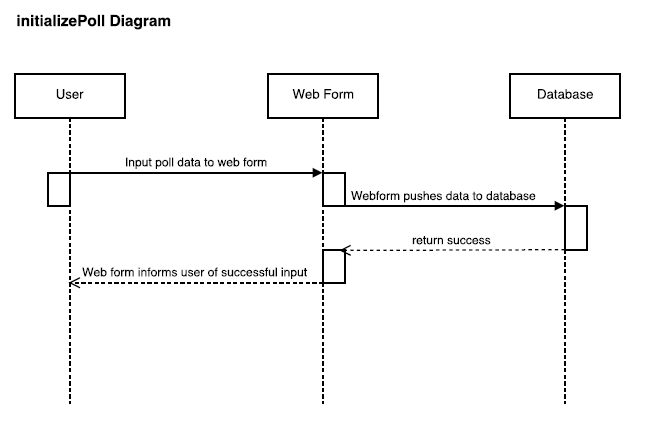
}

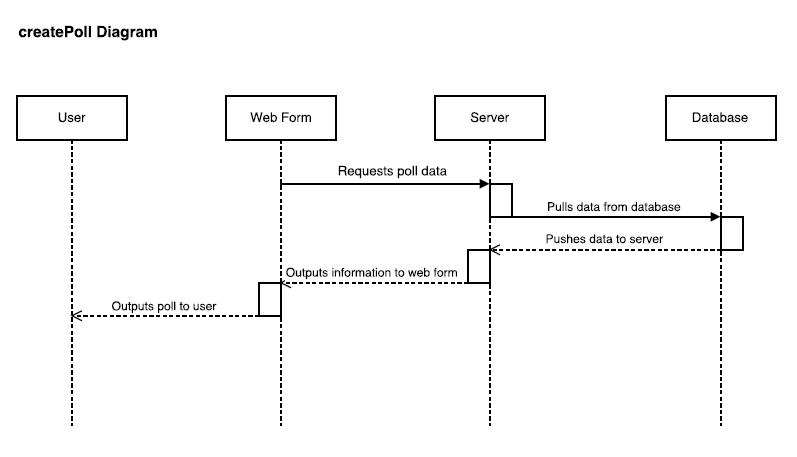
Add 1 to the tally array in location integer points to

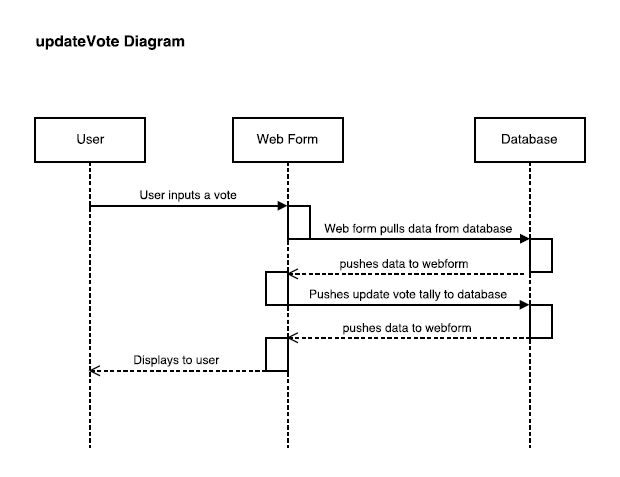
Push data to database

}

Users will have an option to create a pole that other members can vote on. They have to input a title, their question and options to be voted on. The site then creates and displays a poll which users can vote on and see results.







14 profiles:

the profile module contains a users information about themselves such as full name, major, resume,schedule etc.

Signitures:

editName(first,middle,last) - takes in 3 strings to change a users name, multiple middle names must be seperated by a hyphen

editMajor(name) - change the major for the current user

editMinor(name) - change the name of the minor for the current user

addSchedule() - create a schedule object to be filled in with a web form

editSchedule() - open the users current schedule for modification

deleteSchedule() - remove the users current schedule from the profile and database

addResume() - create a resume object to be filled in with a web form

uploadResume(resume file) - create a resume object and fill it with the contents of the given file

editResume() - open the users current resume for modification

deleteresume() - remove the users current resume from the profile and database

editName(first,middle,last){

profile.firstName = first;

profile.middleName = middle;

profile.lastName = last;

}

editMajor(name){

profile.Major = name;

}

editMinor(name){

profile.Minor = name;

}

addSchedule(){

open web form for schedule

wait for user to select save

save input into schedule object

profile.add(schedule)

}

editSchedule(){

open web form for schedule containing the profile’s schedule

wait for user to select save

save web form to schedule object

profile.add(schedule)

}

deleteSchedule() {

profile.remove(schedule)

}

addResume(){

open web form for resume

wait for user to select save

save input into resume object

profile.add(resume)

}

uploadResume(resume file) {

read information from resume file into web form for resume

wait for user to select save

save input into resume object

profile.add(resume)

}

editResume(){

open web form for Resume containing the profile’s Resume

wait for user to select save

save web form to resume object

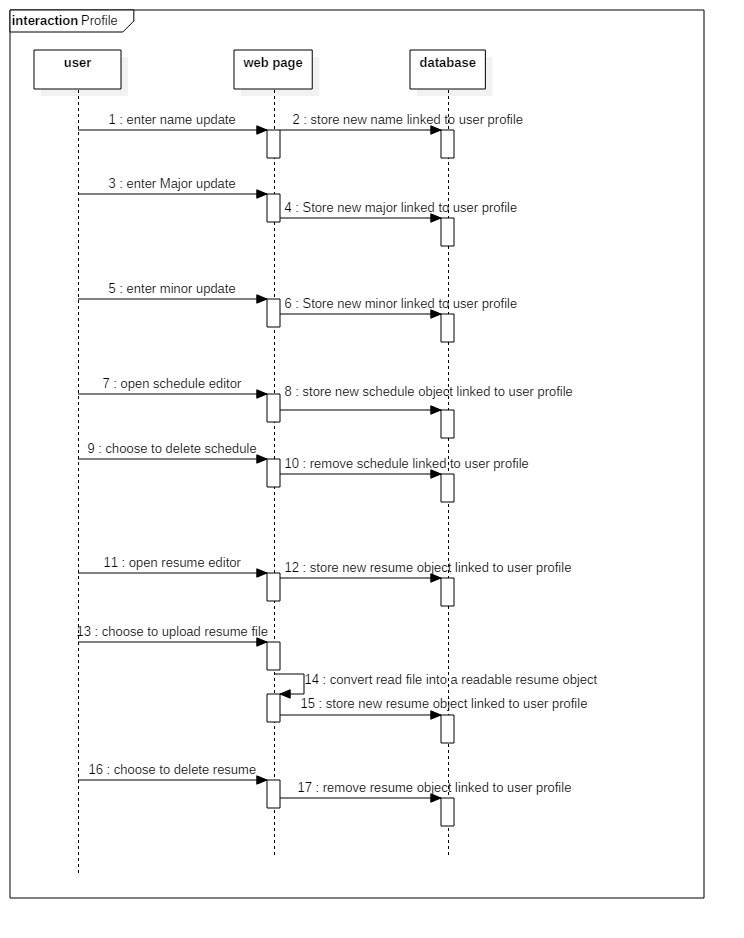
profile.add(resume)

}

deleteResume() {

profile.remove(resume)

}



15 Accordion/Display Photo(Static left part of profiles):

Signatures

loadDisplayPhoto(int id) - when a profile is opened, its display photo is found and displayed

editDisplayPhoto(int id, file input\_ jpgPhoto) - a profiles display photo is adjusted and replaced with this function

deleteDisplayPhoto(int id) - function to delete display photo, there is an x button to trigger this function right under the photo, accessible to the owner only.

loadAccordion(int id) - When a profile is loaded, accordion relevant to profile is found in db and displayed

openAccordionTab(int tabID) - when the user presses on a tab this function opens it, or closes an already open one.

adjustTabContent(int tabID, type content) - if no content is in a tab, it is added/adjusted with this function

This module refers to the non scrolling left side of profiles/groups.

When a profile is opened (or group), loadDisplayPhoto(..) and loadAccordion(..) are called to populate this section. Under the display photo box, a + button is available which will let the user upload a photo fitting our file requirements, which calls editDisplayPhoto(..) using this photo.

When a user opens a tab from the accordion, its content is displayed, if there is no content, a default placeholder image/text will be stored there instead, saying “this user has not added this yet”.

There will be a (+) button available within the tab, to replace your own or the default content with new.

This will call adjustTabContent(..) with the users new content.

void loadDisplayPhoto(int id) {

get profile associated with the id from database

if(profile has display photo)

{

scale jpg to photo box sizing

display it

}

else {

display default anonymous photo

}

void editDisplayPhoto(int id, file input\_ jpgPhoto) {

get profile associated with the id from database

if(file selected meets file requirements){

profile.photo = jpgPhoto

loadDisplayPhoto(int id)

}

else {

let user know to select a valid photo

}

}

void deleteDisplayPhoto(int id) {

get profile associated with the id from database

delete profile.photo from db

loadDisplayPhto(int id)

}

void loadAccordion(int id) {

get profile associated with the id from database

display an accordion with relevant tabs // for example, a persons profile would have a resume but a group wouldn’t

}

void openAccordionTab(int id, int tabID) {

get profile associated with the id from database

if(tabID is already open) {

close tab with tabID

exit function

}

else {

close open tab if there is one

open tab with tabID

display content and scale tab to fit content

}

}

##Some of these tabs would be using the functions/database connections from Profiles/Groups, as that is content stored in some tabs.

void adjustTabContent(int id, int tabID, type newContent) {

get profile associated with the id from database

if(tabContentType == tex and newContent == text within spec) {

tabID.content = newContent

}

if(tabContentType == image and newContent == image within spec) {

tabID.content = newContent

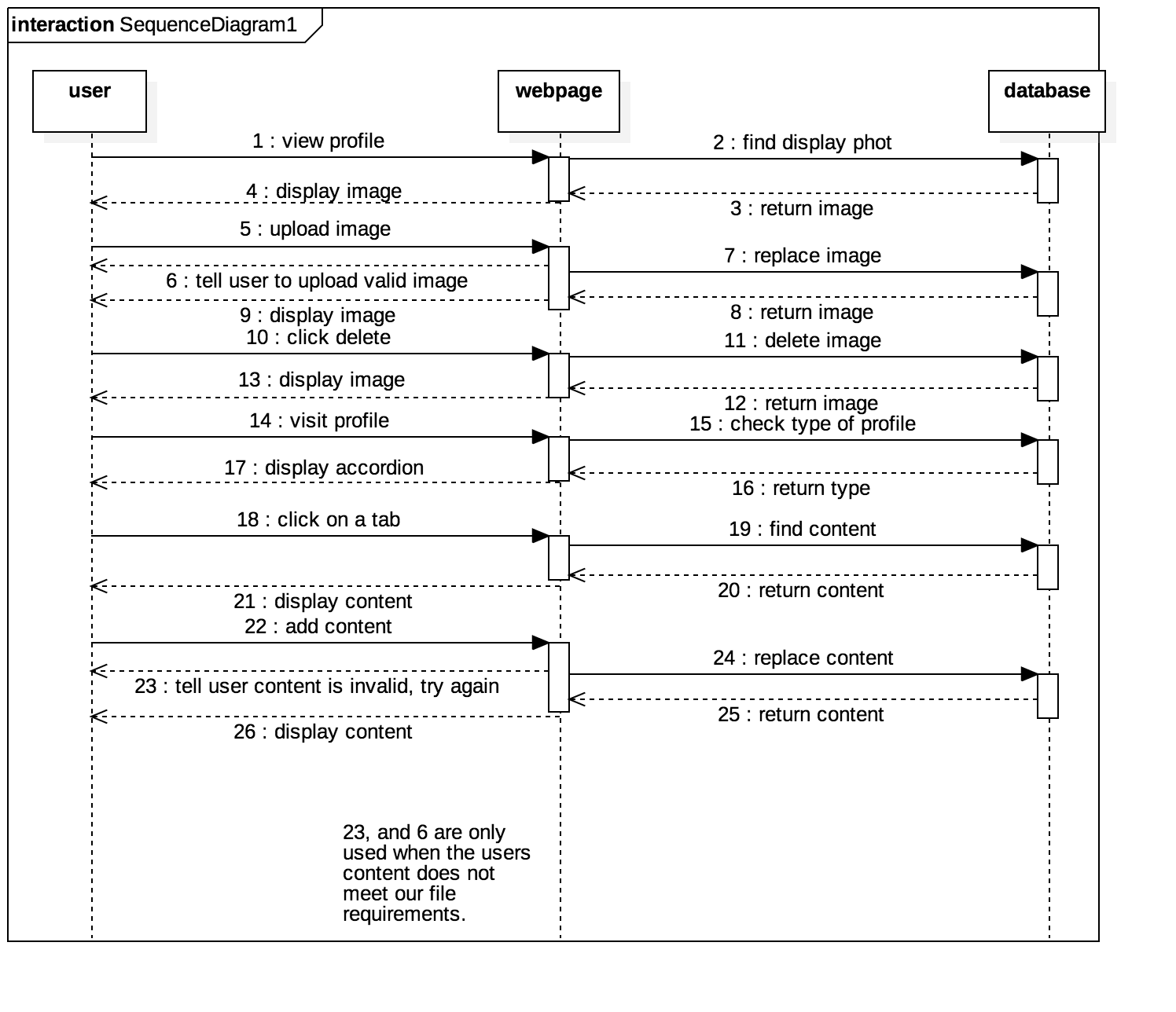
}

else {

print “file does not meet requirements for this tab, please try again

}

}



16 Signatures:

loadComments(int postId) - When a page with posts is visited, this function is called which will load and display all comments

postComment(int postId, int studentId, String comment) - When a user has typed a comment and presses enter, this function is called taking in the postId to make sure the comment is under the indented post, studentId so the comment the user is identified next to the comment, and the comment itself as a string.

editComment(int postId, int commentId, string comment) - A users own post would be edited using this function.

deleteComment (int postId, int commentId) - A user may press the [x] button at the corner of their comment to delete it

viewEdits(int postId, int commentId) - A user may press a back or forward button that comes up on edited comments, which which will adjust the comment to older or newer versions

Anyone with access to seeing a particular post, will also be able to view comments or post, reply and edit (their own) comments. Pressing a comment/reply button will open a box where they can enter a string, and upon pressing enter, postComment will be called with that information (including wether its a reply or not) The comment/reply button would only be available or posts or top level comments, similar to the reply system used on “Facebook”. Clicking the edit button available next too a users own comments, will allow the user to enter text (the current comment by default, filling that text box).

When the user presses enter here editComment(..) is called. An [x] button would be available to the owner of a comment which would called deleteComment(..).

Users can check previously edited versions of comments(including replies) by pressing a left arrow, and newer versions of the comment with the right arrow, both using the viewEdits(..) function. By default on loading a page the most recent comment is always shown.

Currently should be linked to the posts they are on.

void loadComments(int postId, int i) { //int i is optional value

find relevant post object from database using postId

if(comments[] is not empty) {

pull comment objects from database and display under post

display history[0] } //newest edit

}

}

//this function will be called when pages with posts are loaded

void postComment(int postId, int studentId, String comment, boolean isReply) {

if(comment =! an empty string) {

create comment object with a unique id, a postId to connect it to the relevant post, the string comment which will be displayed, in an array called history[], and a boolean isReply to represent if its a top level comment or a reply.

call loadComments(..) to update

}

}

void deleteComment (int postId, int commentId) {

get the relevant comment object from database using postId and commentId and delete it from db.

call loadComments(..) to update.

void editComment(int postId, int commentId, string comment) {

get the relevant comment object from database using postId and commentId

if (comment is == to current comment at the front of the history[] array) {

do nothing, we don’t need duplicate comments in a row, in the edit history

}

if(comment is an empty string) {

call deleteComment(..)

call loadComments(..)

}

else {

add string comment to the front of history[]

call loadComments(..)

}

}

viewEdits(int postId, int commentId, bool goBack) {

get the relevant comment object from database using postId and commentId

check which comment is currently displayed from history[], and take that corresponding integer

and set i = to it.

if(goBack) {

if(history[i + 1] exists) {

pull from database and display history[i+1] for the current comment

}

}

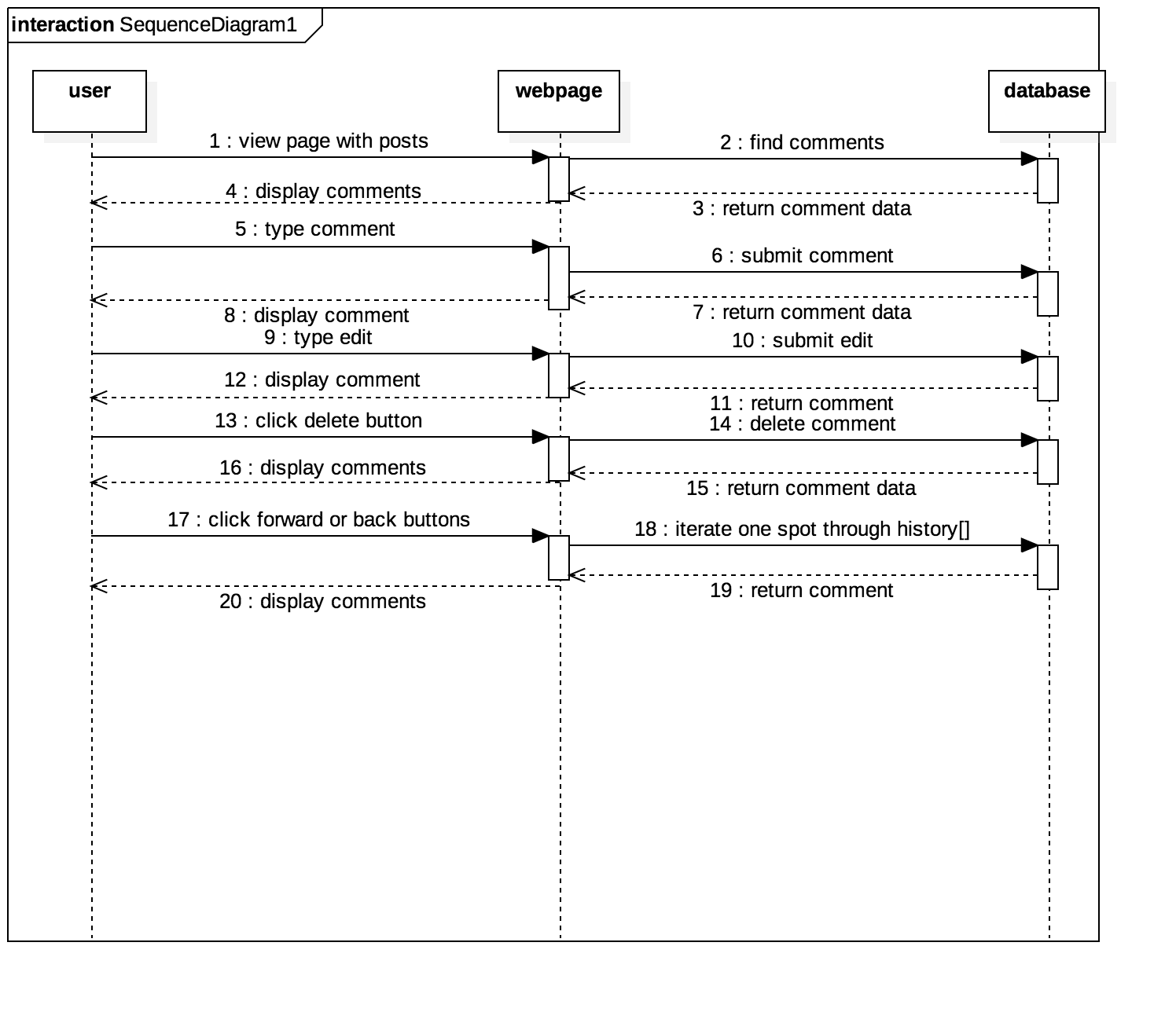
if(!goBack) { //so going forward)

if(history[i - 1] exists) {

pull from database and display history[i-1] for the current comment

}

}



Nav Bar

Signatures:

returnToSelf(int id) - called when the user presses “home” button

search(String goal) - used to search our database for a specific query

The nav bar, static at the top of the page, includes a home button at the top left which calls returnToSelf(int id) for the current users id and redirects them to their profile. In the middle is a search bar, which when typed in, and enter is pressed, calls the function search(..) which will make a search in the DB, and redirect the user to a results page which displays the search results

void returnToSelf(int id) {

query db for users id, and redirect user to that profiles webpage

]

void search(String goal) {

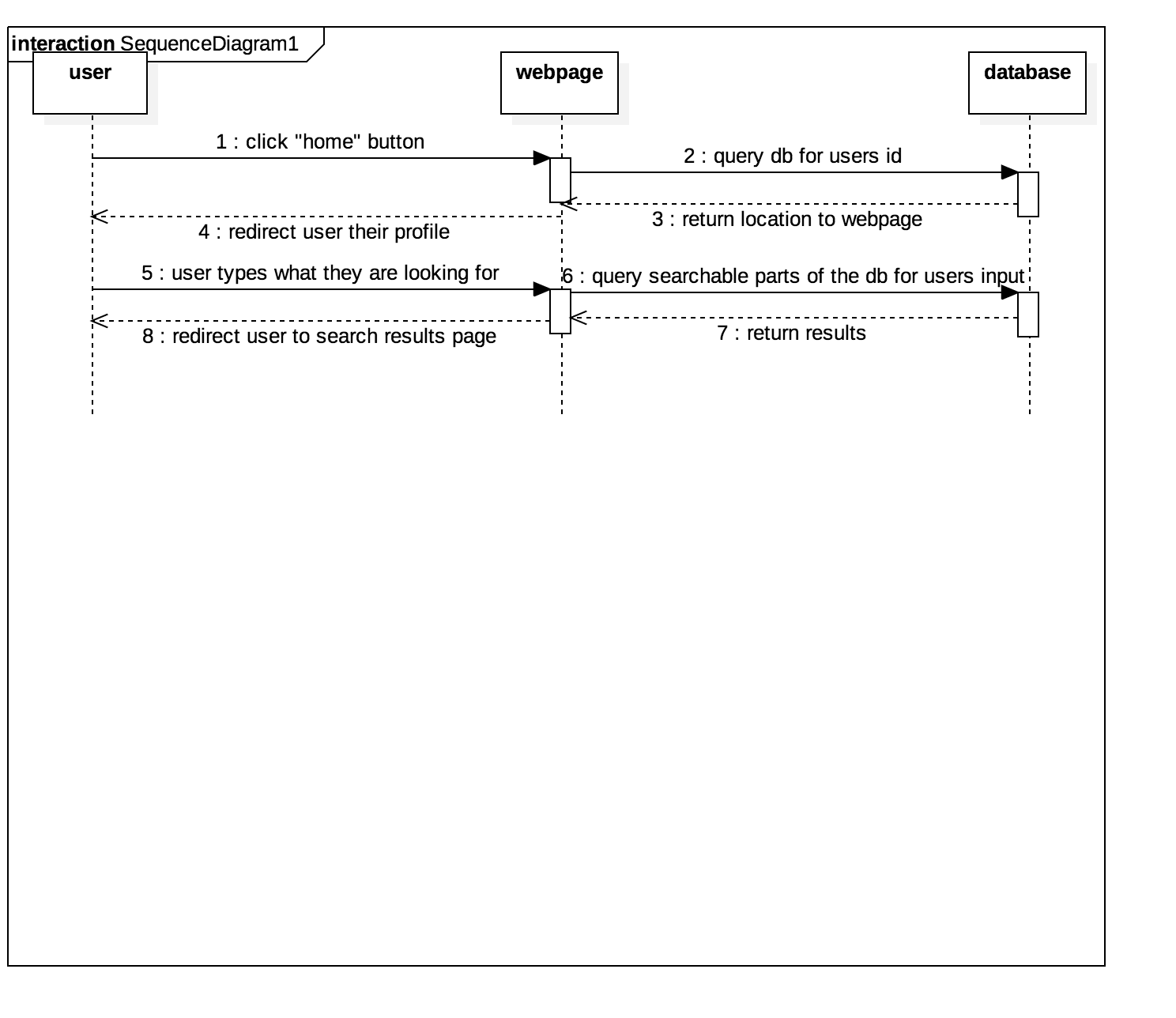
query the searchable parts of the database for items matching the String goal

redirect to page displaying the results

if(query returned no results) {

print “sorry, we couldn’t find anything matching your search, please try again”

}



# Size and Performance

The modular design ensures that the overall system is fast and kept to a small size. As the database grows, so too will database operations. However, by both reducing the number of possible operations and by restraining registrants to MUN students, database size will be kept at a manageable size.

# Quality

By breaking modules into the design presented above, the system remains highly modifiable, while also being secure from one module causing another to break. Data security and stability is also assured by reducing the number of operations that link to the database.