Memorial University of Newfoundland Social Network(MUNSN)

Prepared By: Group D
Tyler Vey
Saahil Budhrani
Mark Hewitt
Allan Collins

MUNSN Architectural Document

February 26, 2017

1.0 Overview

This document lays out the architecture of the MUN Social Network (MUNSN) application. We are using a 3 tier architecture; having a presentation tier (web server), service tier (web service), and a data tier (mongoDB database). The users browser only connects to the presentation tier. The users browser never connects to the web service or the database directly. This improves security and scalability as discussed in the SRS document. Having mongoDB calls in a single place (web service) improves maintainability and enforces re-usability.

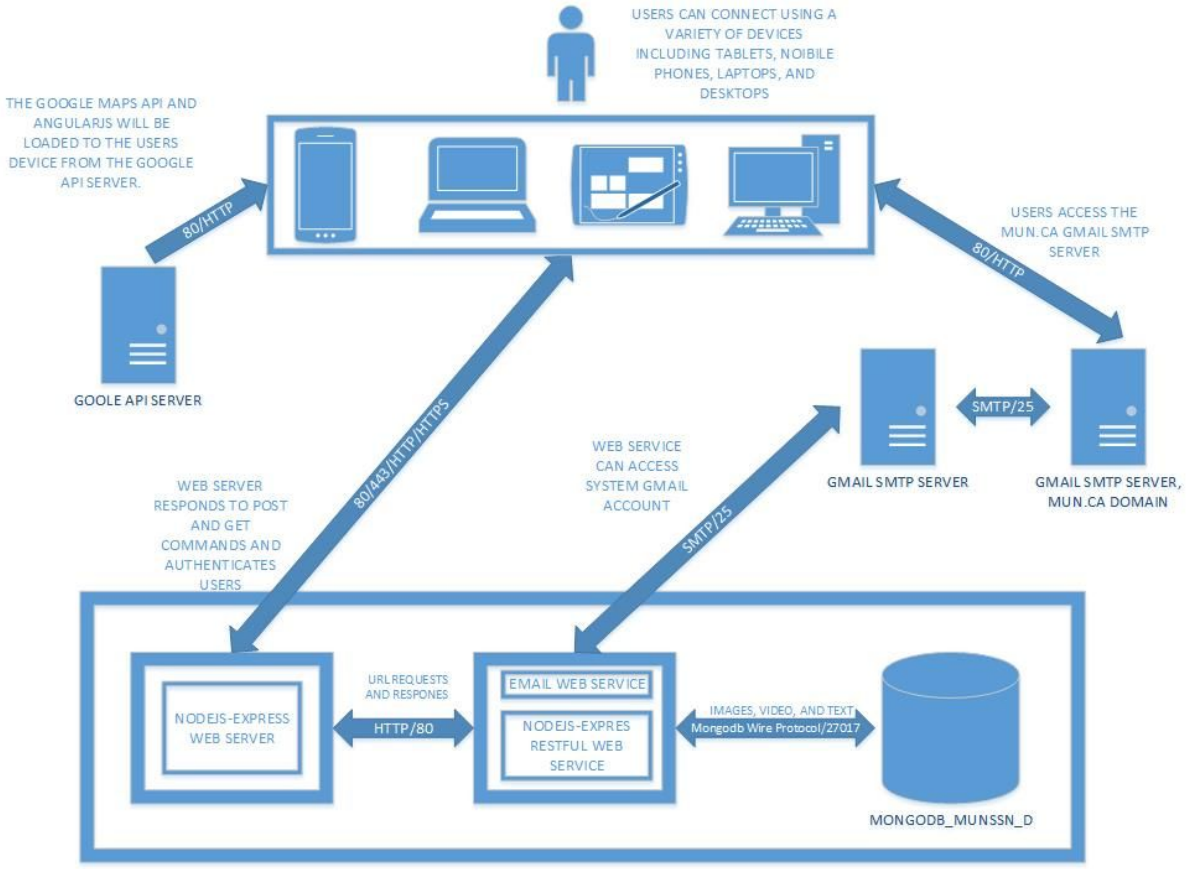
We first describe the hardware components - the three tiers mentioned above - and how they break down into modules and sub-modules. Next we look at how the software is broken down into modules and sub-moduels.

Then we discuss the service calls that the system requires. We list the required inputs and the expected output. This enables the back-end to be developed independently from the front-end. We also see the access patterns and the data flows - from this we call build the mongoDB schema. The mongoDB schema is presented.

Finally we look at how we are dividing the work amongst the team members and how it will be scheduled as well as what the responsibilities were for this current document.

2.0 Hardware Modules

1. NodeJS-Express Web Server
    a. Multiple publisher-subscriber modules
        i. Timeline
        ii. Polls
        iii. Chat
2. NodeJS-Express-Mongoose RESTful web service
3. MongoDB database



3.0 Software Modules

This is a single-page application. Each module is highly cohesive and loosely coupled. A single business transaction only touches a single module. This allows us to let each module have its own client-side MVC model. When we have a case we require information from another module instead of coupling the the two and introducing a dependency we simply have the new module re-use the service call.

For example the Lost and Found Module needs to know if the person who lost an item is their friend so their cell number can be displayed. We could introduce a dependency and couple the Lost and Found Module to the Friends Module but it is much preferable to re-use the service call that returns a user's list of friends. This ensures modules loosely-coupled.

When a new module is loaded the old MVC model is no longer needed - loosely coupled. The controller makes the calls to the web service and changes the model and then the view is updated. Module ideally should be independant or possibly loosely coupled.

We are using good design patterns to keep the modules loosely coupled. For Example by using the publisher-subscriber pattern neither 'Polls', nor 'Study Groups', nor 'Friends' or any other

module need to be coupled to the Timeline module - they simply update the publisher-subscriber on the web server and a new publication is sent to all subscribers.

In keeping with good usability principles every service call results in a change in the user interface so that the user can know that the action has been performed or has failed.

3.1 Software Modules List

1. Timeline Module
2. Resume Module
3. Private Messaging
4. Lost and Found
5. Polls
6. Friends
7. Study Groups
8. Account Module
9. Schedule
10. Web Communications module -see section 3.3 for description

3.2 Server Side Modules

1. RESTful Web Service
    a. Authentication Sub-Module
    b. MongoDB Queries Sub-Module
2. Web Server
    a. Publisher-Subscriber Sub-Module
3. Database

3.3 Client-Side Javascript Sub-Modules

Each software module will have its own client side mvc model and they will all use the communications module to access the connection to the web server and receive and send data. There will be;

- Nine model submodules named ‘model<*SoftwareModuleName.js*>’
- Nine view submodules named ‘view<*SoftwareModuleName.js*>’
- Nine controller classes named ‘controller<*SoftwareModuleName.js>’* all call the WebCommunications module which is a helper function that creates the connection to the web server and handles GET and POST requests. It controls all the flow of data to and from the individual module controllers.
- A main model, view, and controller sub-modules named
    - modelMain
    - viewMain

- controllerMain - aka ‘WebCommunications’. All controllers will use this class to access the connection to the web server and send and receive data.

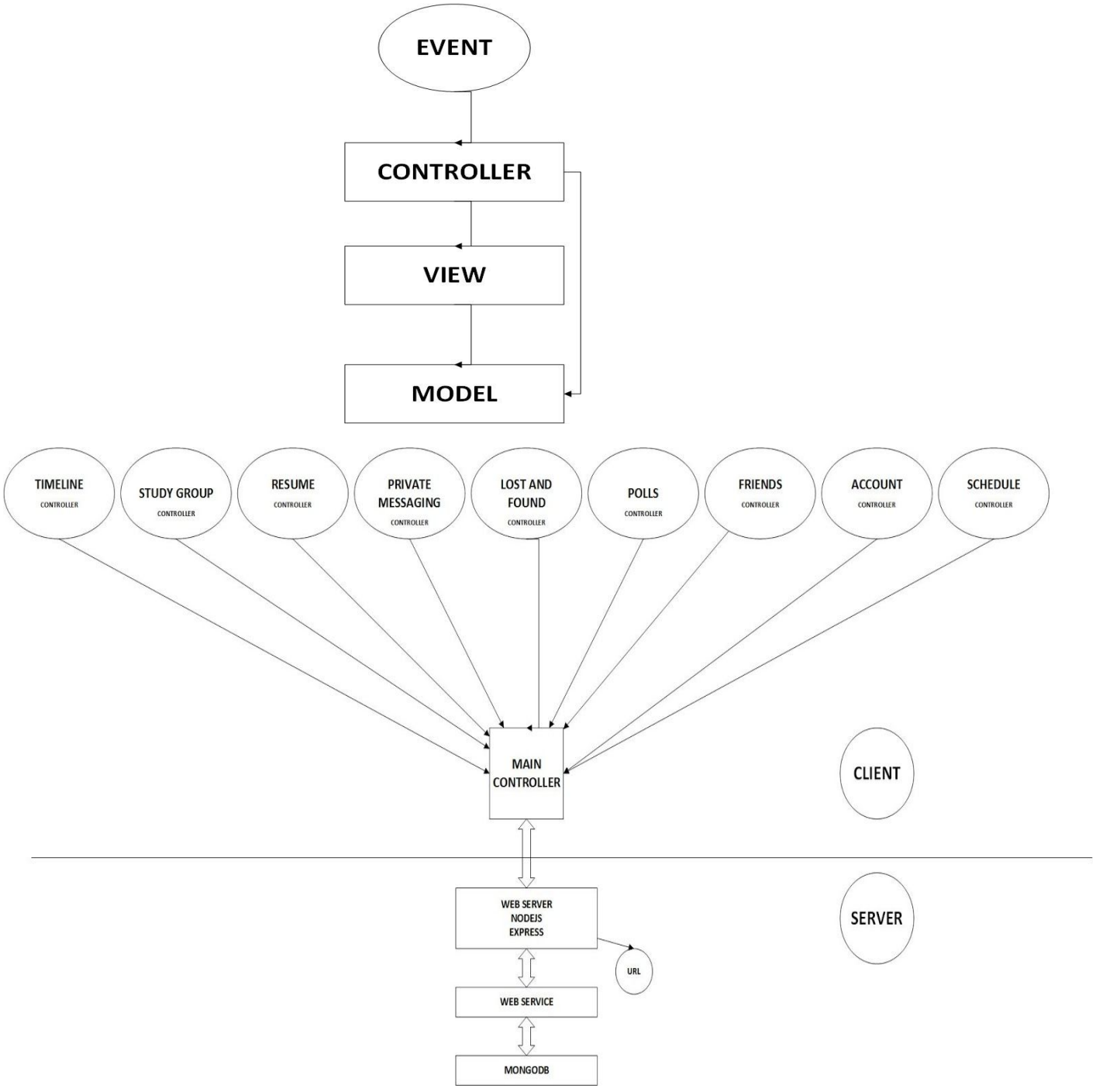


Figure 1 - CLASS DIAGRAMS

Service Calls

Below is the list of service calls that the application will require. This is essential so that backend and front-end developers can work independently and know what the inputs/outputs are. Note that,
GET Requests - URL contains parameters
DELETE Requests - URL contains parameters
POST requests - URL does not contain parameters.

Knowing the list of service calls enables us structure our mongoDB database according to the access pattern for efficiency. It also serves as a roadmap so that front-end and back-end developers are on the same page.

	Module 1: Timeline
SC-TL-01	**View a Timeline**
Input	Parameters: timeline_id, viewer's user_id
Sample URL	http://cs.mun.ca/s77ajc/MUNSN/timeline/view?timeline_id=1234567&user_id=7654321
Output	Content including text, images, video and comments grouped by post. Posting Rights of the Viewer for each post.
Sample JSON	{ "Posts": [{ "post_id":"<post id>", "user_id":"<user of timeline owner> "poster":"<poster id>", "post_time":"<date timestamp>", "post_text":"<post text>", "media_url":"<media url>", "Posting_right":"1", "Comments": [{ "user_id":"<user id of commenter>", "comment_text":"<comment text>", "date_time":"<date time stamp>"

	}, { “user_id”:”<user id of commenter>”, “comment_text”:”<comment text>”, “date_time”:”<date time stamp>” }] }, { “post_id”:”<poster id of second poster”, ……. }] }
Explanation	Determine if the viewer is the owner of the timeline and if not whether this viewer has visibility or posting rights. Visibility is determined by the web service and only posts which the user is allowed to see will be returned. If it is not the viewers timeline then the timeline options are not available. The timeline may not be visible to the viewer or the viewer may not be able to post. The User is subscribed to the publisher-subscriber model for this timeline.
Web-Server Request Type	GET

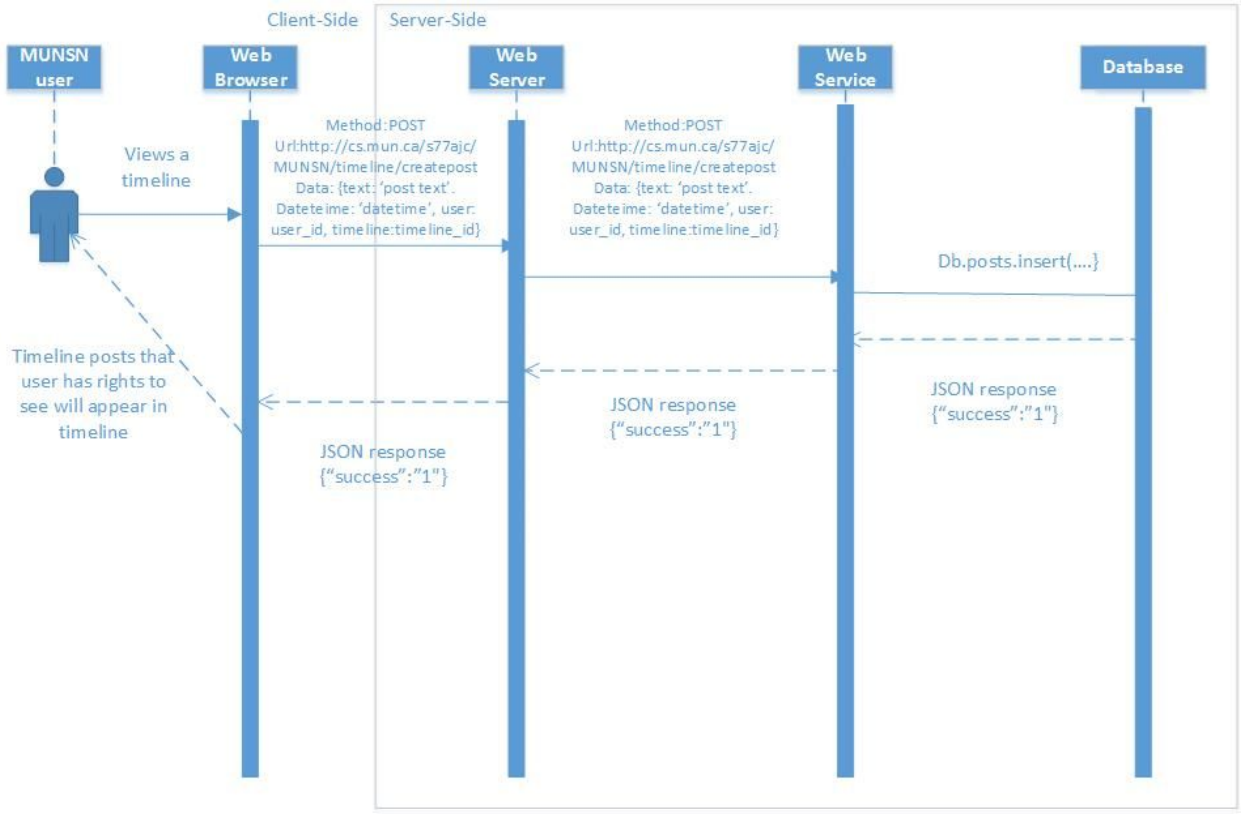


Figure SC-TL-01

	Module 1: Timeline
SC-TL-02	**Create a Post**
Input	Parameters: poster_userid, timeline_id Data: Post text, optionally media.
Sample URL	http://cs.mun.ca/s77ajc/MUNSN/timeline/createpost
Output	1 or 0. Verification that data has been loaded into db. So client side MVC model can be updated.
Sample JSON	{"success":"<1 or 0>"}
Explanation	Can determine who is posting and where they are posting and if they have rights to post. Verify that they have rights to post. POST request contains text and optionally media. Publisher-Subscriber model for this timeline is updated and a new publication is pushed out to all subscribers. The new post will show without the page being refreshed.
Web-Server Request Type	POST

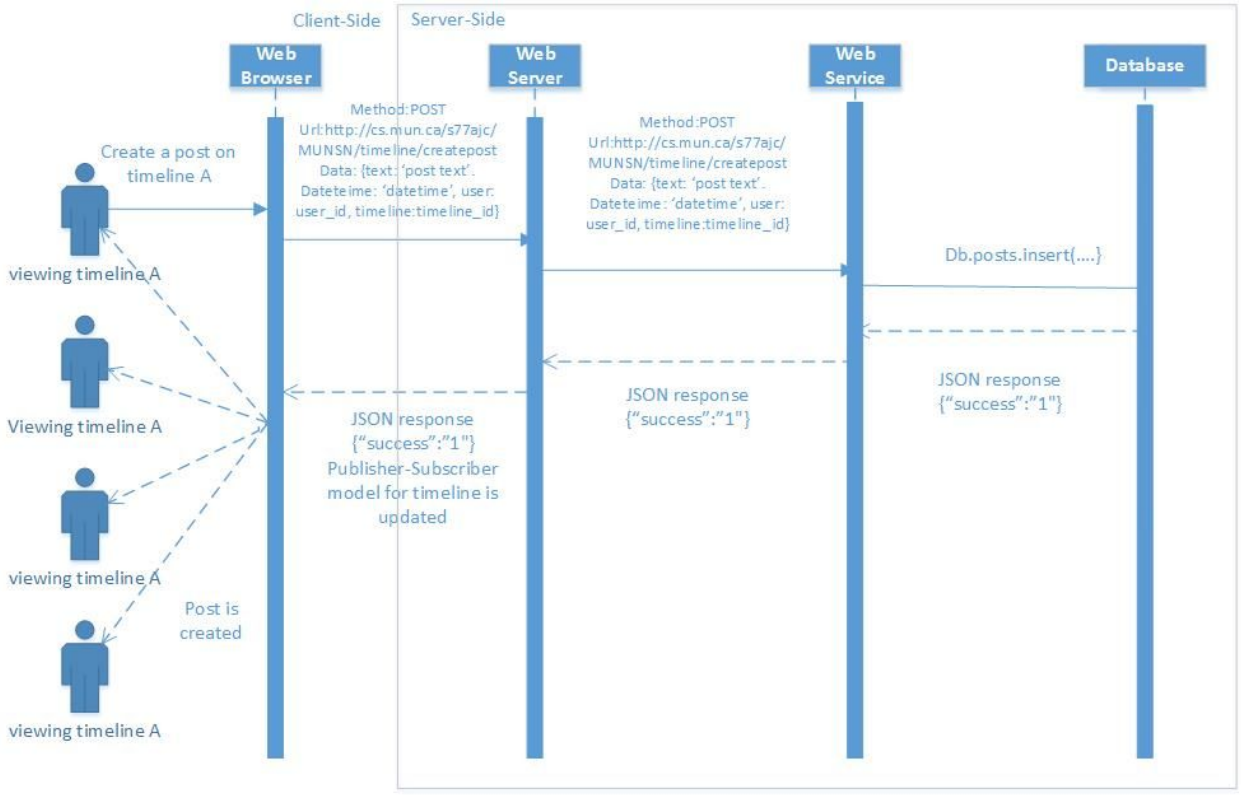


Figure SC-TL-02

	Module 1: Timeline
SC-TL-03	**Comment on a Post**
Input	Parameters: user_id of commenter, timeline_id Data: comment text
Sample URL	http://cs.mun.ca/s77ajc/MUNSN/timeline/comment
Output	Text Comment. Verification that text has been uploaded.
Sample JSON	{"success":"<1 or 0>"}
Explanation	Can determine if the user has rights to post comments on this timeline. Have comment show up under post when verified it has been loaded in database. The publisher-subscriber model for this timeline is updated and a new publication is pushed out to all subscribers. The new comment will show without the page being refreshed to all viewers of this timeline.
Web-Server Request Type	POST

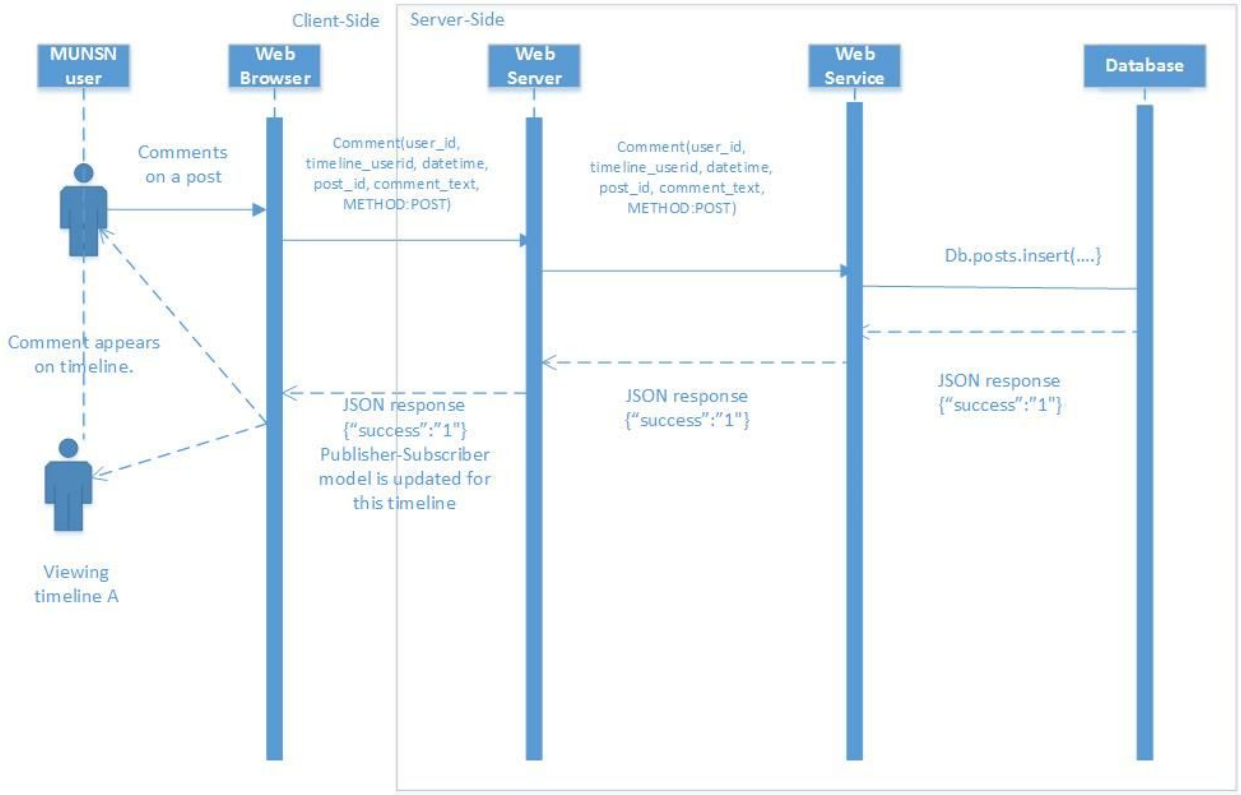


Figure SC-TL-03

	Module 1: Timeline
SC-TL-04	**Edit a Comment**
Input	Parameters: user_id of editor, post_id, comment_id, datetime Data: comment text
Sample URL	http://cs.mun.ca/s77ajc/MUNSN/timeline/editcomment
Output	Confirmation that the comment has been edited.
Sample JSON	{"success":"<1 or 0>"}
Explanation	Determine what timeline(post_id), what comment(comment_id), when(datetime), are they allowed(user_id of editor), and what(comment text)
Web-Server Request Type	POST

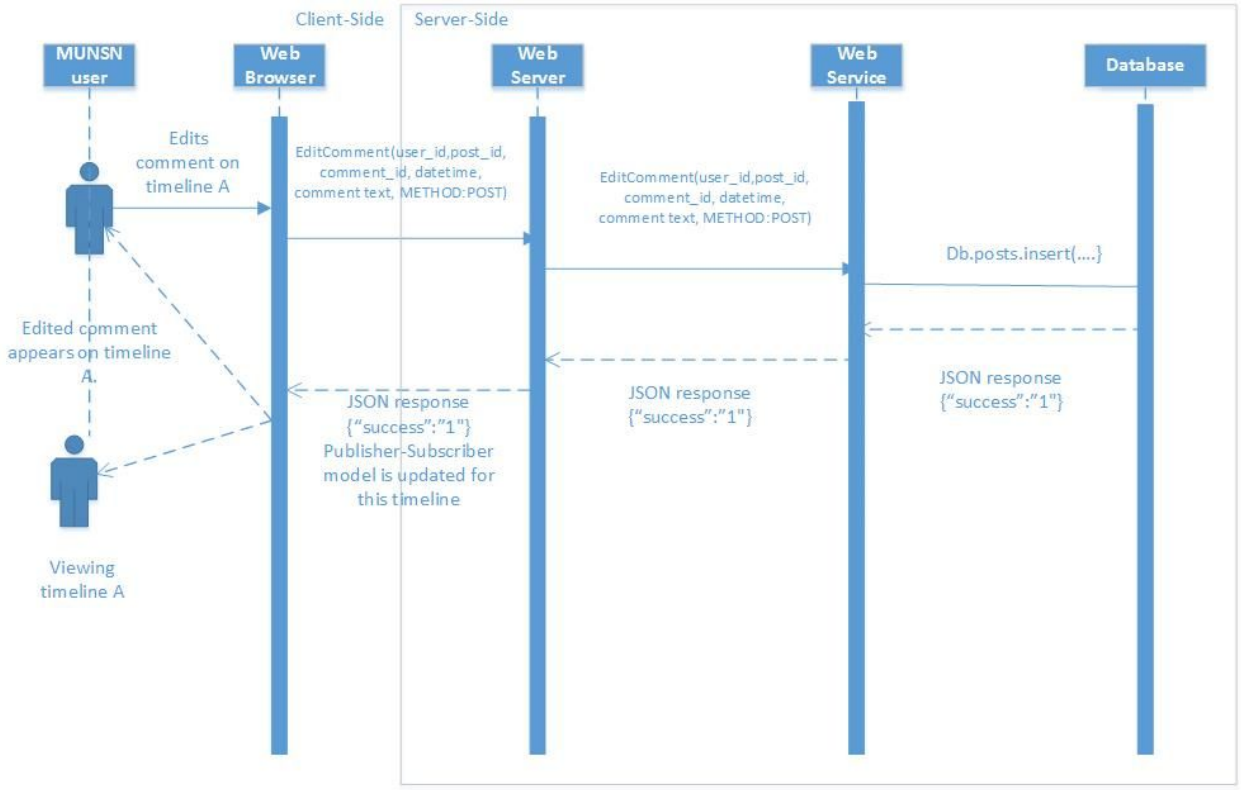


FIgure SC-TL-04

	Module 1: Timeline
SC-TL-05	**View Comment Edit History**
Input	Parameters: post_id, comment_id, user_id of viewer
Sample URL	http://cs.mun.ca/s77ajc/MUNSN/timeline/commentedithistory?post_id=1234&comment_id=4321&user_id=5555
Output	Confirmation that comment has been edited.
Sample JSON	{"success":"<1 or 0>"
Explanation	Post_id identifies the post, comment_id identifies the comment, and user_id determines if the user if allowed to edit.
Web-Server Request Type	GET

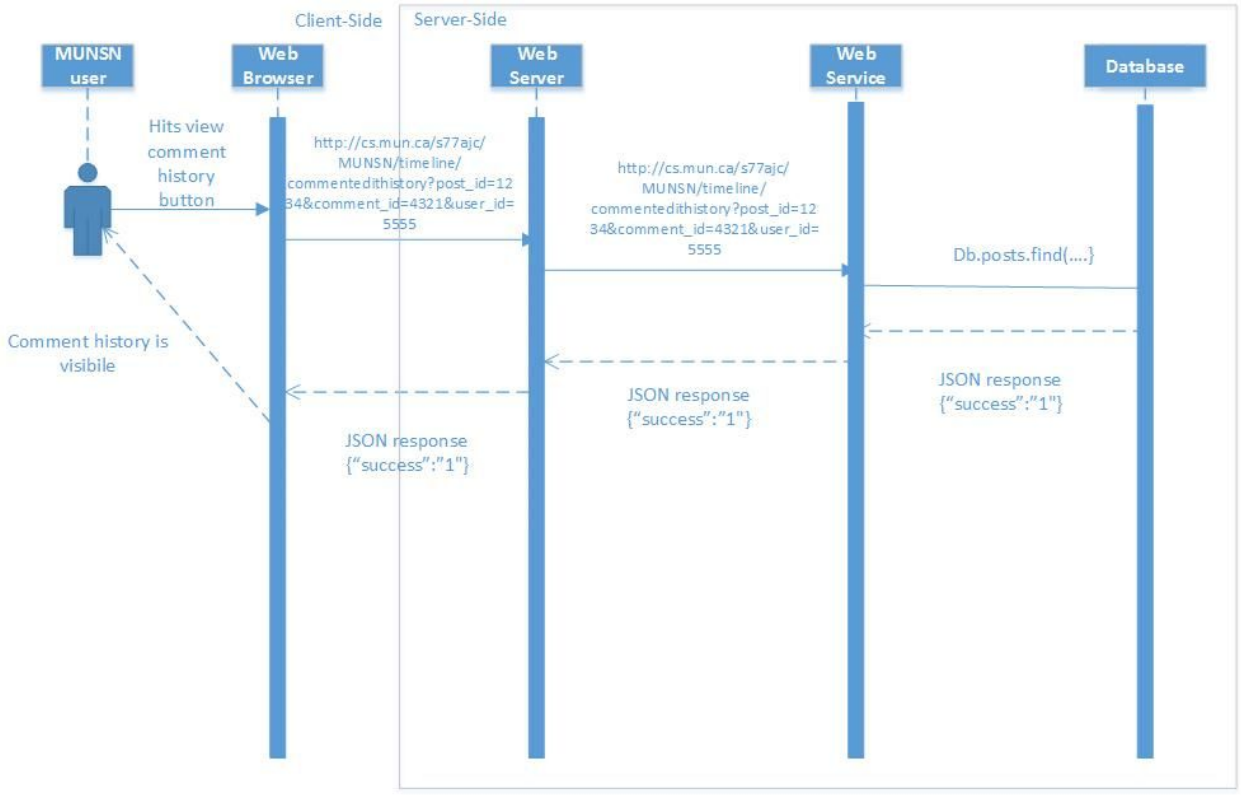


Figure SC-TL-05

	Module 1: Timeline
SC-TL-06	**Reply to a Comment**
Input	Parameters: post_id, comment_id, user_id Data: Comment reply text
Sample URL	http://cs.mun.ca/s77ajc/MUNSN/timeline/replycomment
Output	1 or 0 to determine if the comment reply was successfully posted to the database.
Sample JSON	{"success":"<1 or 0>"
Explanation	Post_id determines which post, comment_id determines which comment and user_id determines if the user is allowed to reply to comments on this timeline. Reply text is sent with the POST request.
Web Server Request Type	POST

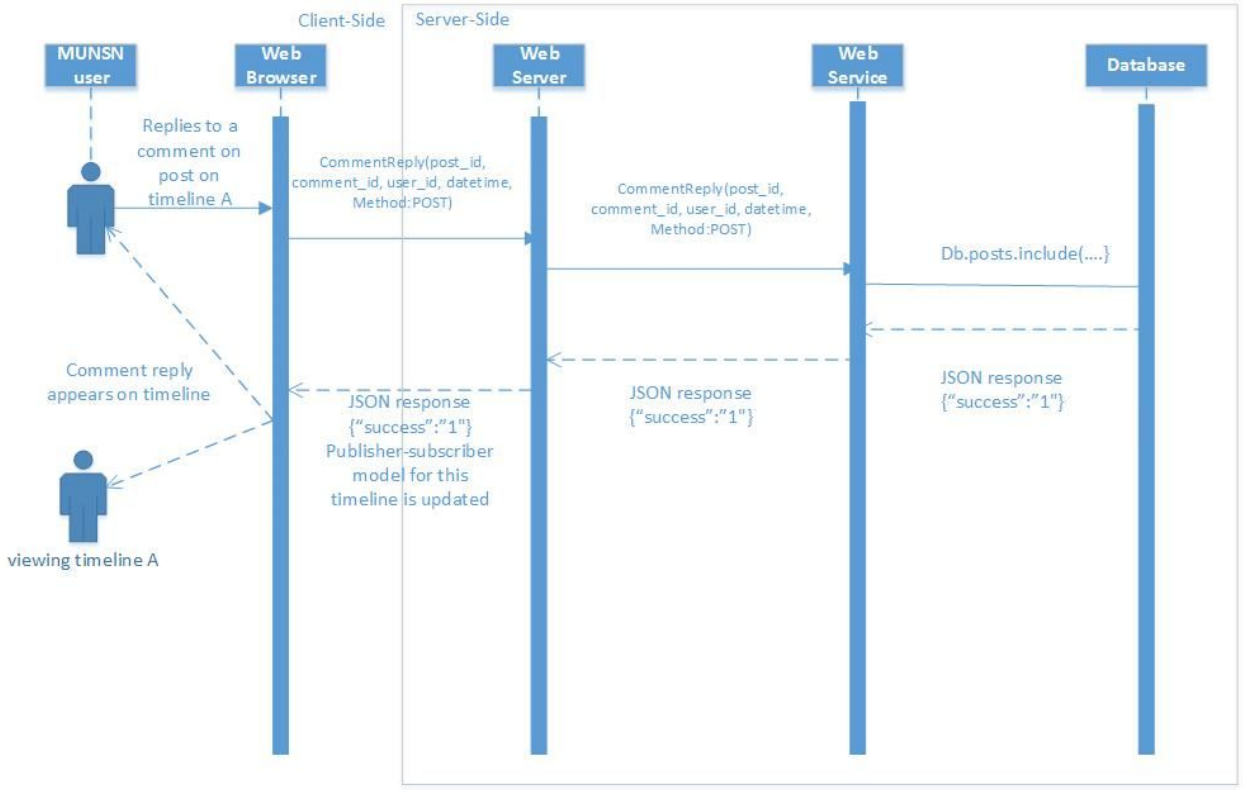


Figure SC-TL-06

	Module 1: Timeline
SC-TL-07	**Change Timeline Posting Rights**
Input	Parameters: userid Data: posting_right*, optionally a list of friends if 'List' is chosen *One of 'Everyone', 'User', 'Friends', 'List'.
Sample URL	http://cs.mun.ca/s77ajc/MUNSN/timeline/changeTimelinePostingRights
Output	Verification that change has been applied to mongoDB database
Sample JSON	{"success":"1"}
Explanation	Have user interface let user know that the change to posting rights has been applied. If 'List' is chosen then the database would determine the list of friends from the parameter userid. POST request contains the type of change. Update the publisher-subscriber model for this timeline and push out a new publication to all subscribers. Posting rights will change for all affected viewers without a page refresh.
Web-Server	POST

Request Type	

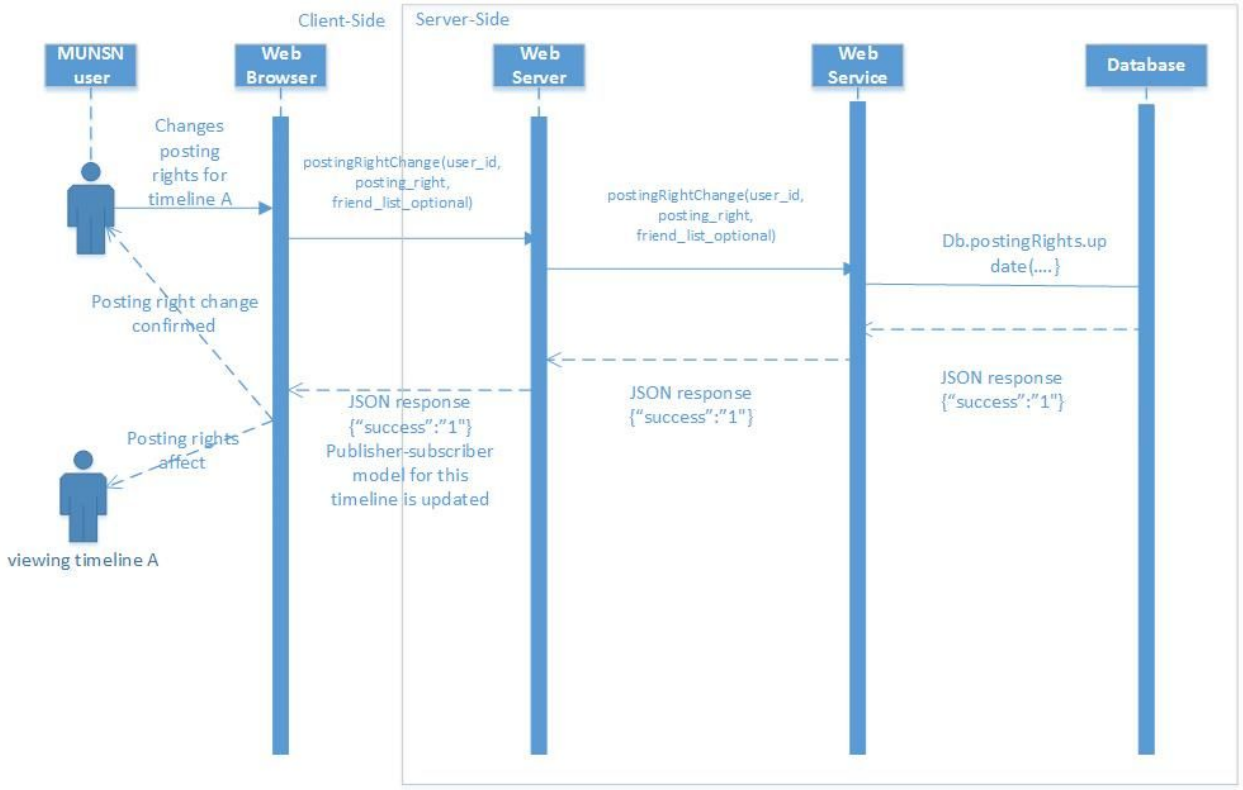
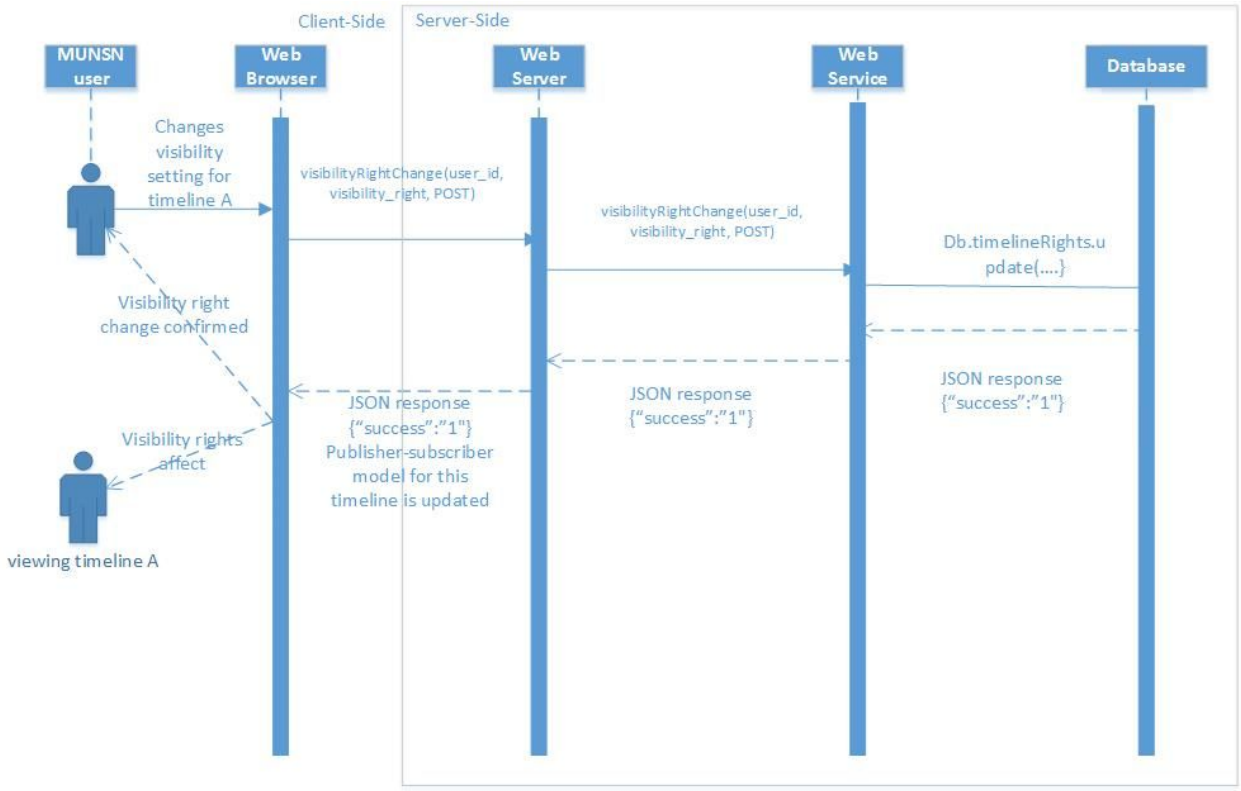


Figure SC-TL-07 - Timeline posting right change

	Module 1: Timeline
SC-TL-08	**Change Timeline Visibility**
Input	Parameters: Userid Data: Visibility* Visibility options include one of Private, Friends, Public.
Sample URL	http://cs.mun.ca/s77ajc/MUNSN/timeline/changeTimelineVisibility
Output	A private, friends only or public timeline.
Sample JSON	{"success":"<1 or 0>"}
Explanation	Depending on the Timeline visibility the output can be either a private, friends only or public timeline. User is notified that the change has been applied when the positive reply from the web server is received. The publisher-subscriber model is updated and a new publication is pushed out to all subscribers. Posts disappear or appear for all affected viewers without a page refresh.

Web-Server Request Type	POST



SC-TL-08 - Timeline visibility change

	Module 1: Timeline
SC-TL-09	**Change Timeline Individual Visibility**
Input	Parameters: timeline_id, post_id Data: Visibility Selection
Sample URL	http://cs.mun.ca/s77ajc/MUNSN/timeline/changeIndividualVisibility
Output	A Post that is visible to one of Private, Friends, Public depending on the user's choice. The publisher-subscriber model is updated for this timeline and a new publication is pushed out to all subscribers. Change is applied to all viewers of this timeline without a page refresh.
Sample JSON	{"success":"<1 or 0>"
Explanation	Can determine which post and timeline to apply the visibility change.
Web-Server Request Type	POST

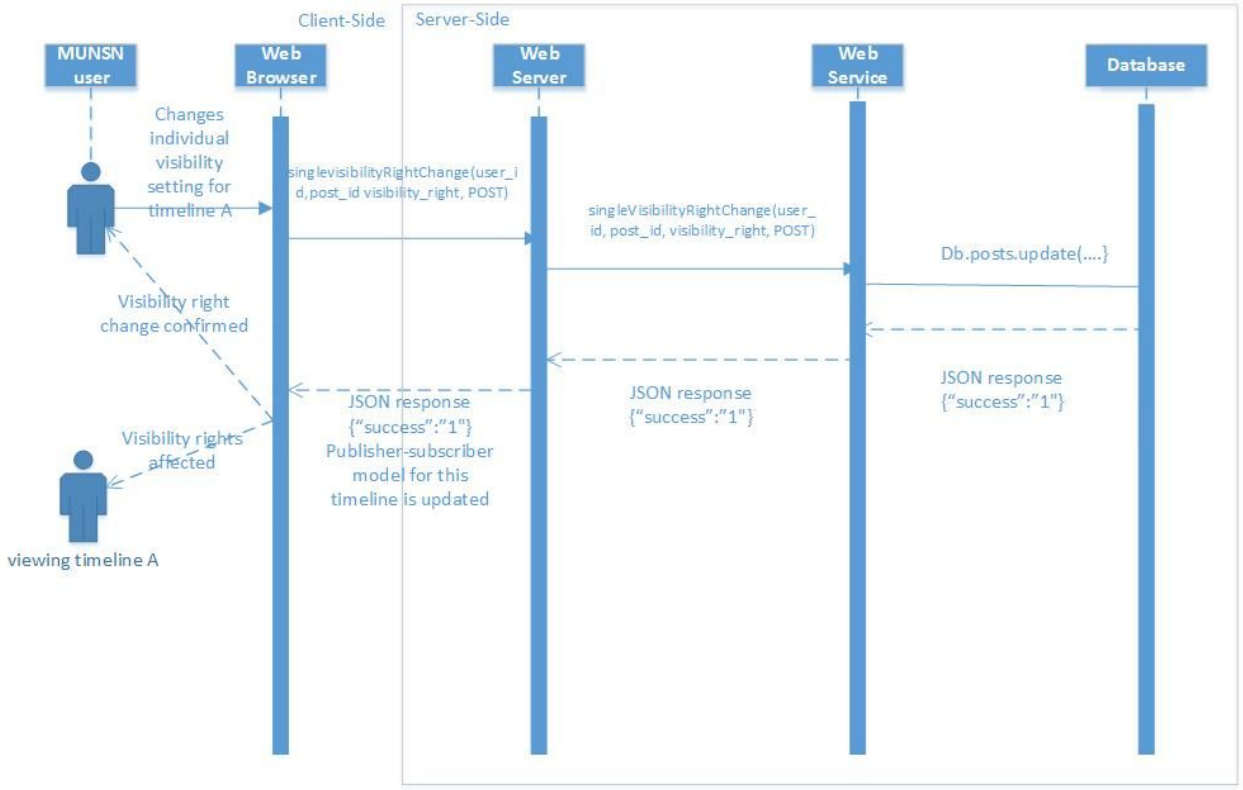


Figure SC-TL-09 - Change Individual Visibility for a timeline post.

	Module 2: Study Group
SC-SG-01	**CREATE A STUDY GROUP**
Input	UserID of Group creator
Sample URL	http://cs.mun.ca/s77ajc/MUNSN/userID/studygroup01/
Output	A group in which members can be added or invited by the admin or selected members.
Sample JSON	POST http://{hostname}:{port} /groups/?{parameters} Accept: application/json Content-Type: application/json { "id": "97447ea3-a95f-4d29-ba6e-d65fc2e84e85", "name": "Example Group", "enabled": true, "users": [{ "id": "00000000-0000-0000-0000-000000000003",

	"name": "releaser", "displayName": "releaser" }, { "id": "00000000-0000-0000-0000-000000000002", "name": "admin", "displayName": "admin" }
Explanation	The user creates a study group and sets the privacy of it to ‘private’ or ‘public’
Web-Server Request Type	POST

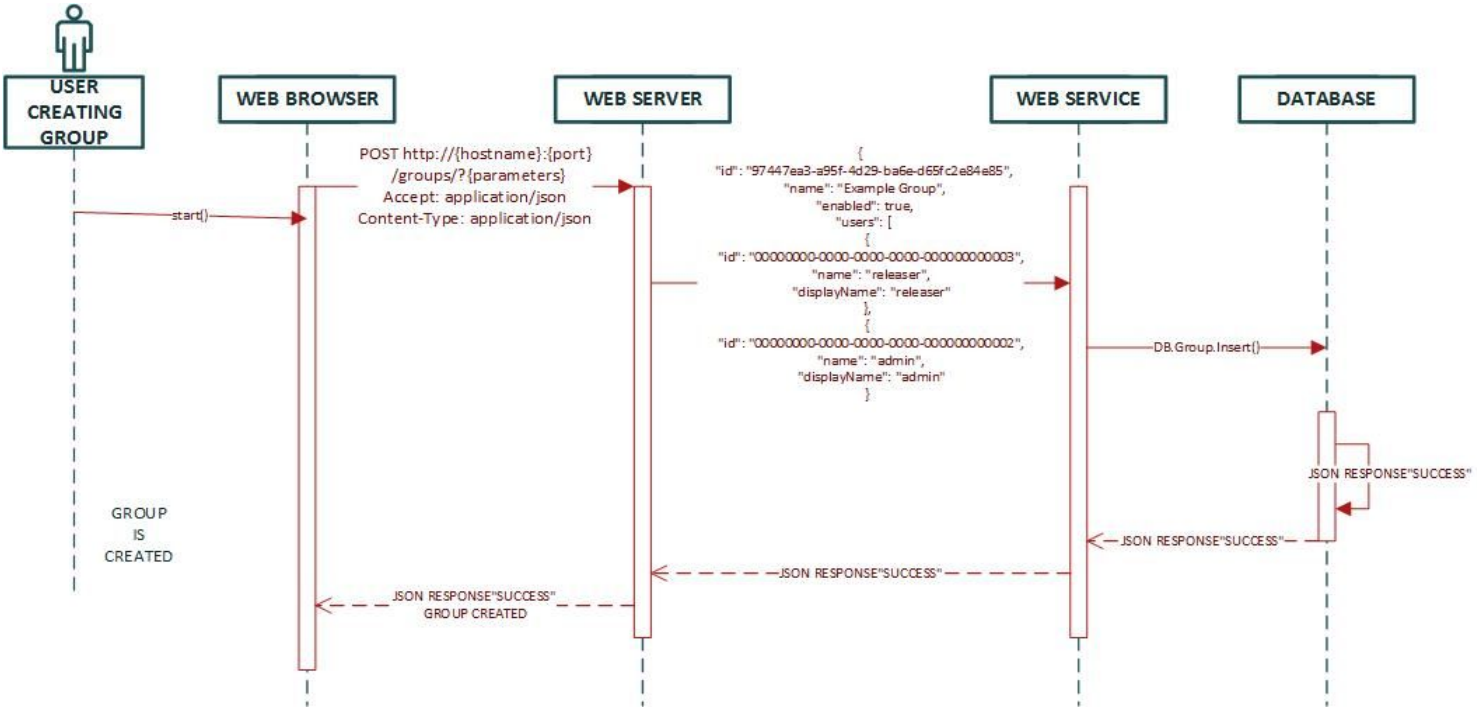


Figure SC-SG-01 - CREATE STUDY GROUP

SC-SG-02	**INVITE TO STUDY GROUP**
Input	UserID of Admin and invited members
Sample URL	http://cs.mun.ca/s77ajc/MUNSN/userID/studygroup01/inviterequest
Output	Members will be added into a group.

Sample JSON	{ "kind": "directory#member", "id": "group member's unique ID", "email": "liz@mun.ca", "role": "MEMBER", "type": "GROUP" }
Explanation	Members will be added into a group automatically if it is a public group or members will have to accept a group invitation to be added into the group if it is a private group.
Web-Server Request Type	POST

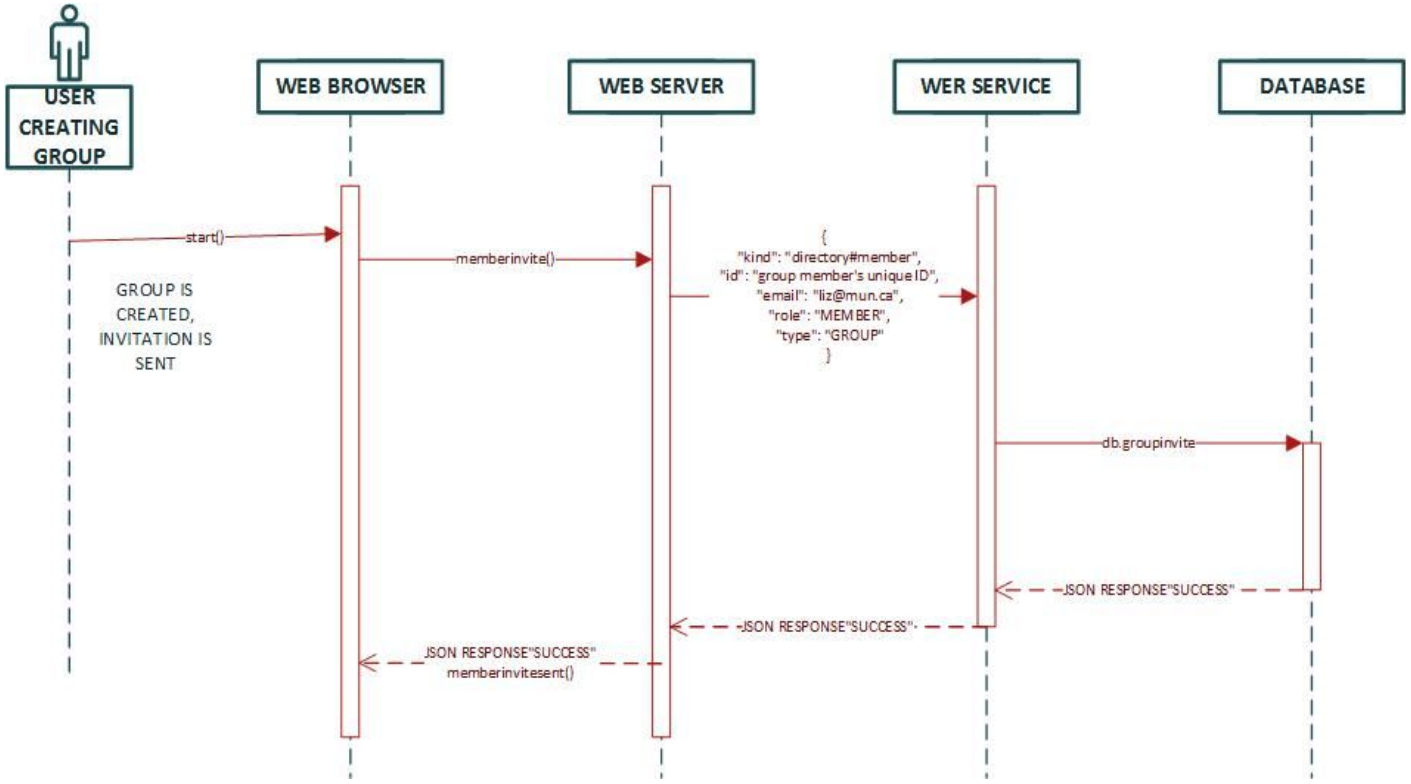


Figure SC-SG-02- INVITE TO STUDY GROUP

SC-SG-03	**ACCEPT STUDY GROUP INVITATION**

Input	UserID of Admin and invited members
Sample URL	http://cs.mun.ca/s77ajc/MUNSN/userID/studygroup01/acceptrequest
Output	Members will be accept invitation into a group.
Sample JSON	{ "kind": "directory#member", "id": "group member's unique ID", "email": "liz@mun.ca", "role": "MEMBER", "type": "GROUP" }
Explanation	Members will be added into a group automatically if it is a public group or members will have to accept a group invitation to be added into the group if it is a private group.
Web-Server Request Type	POST

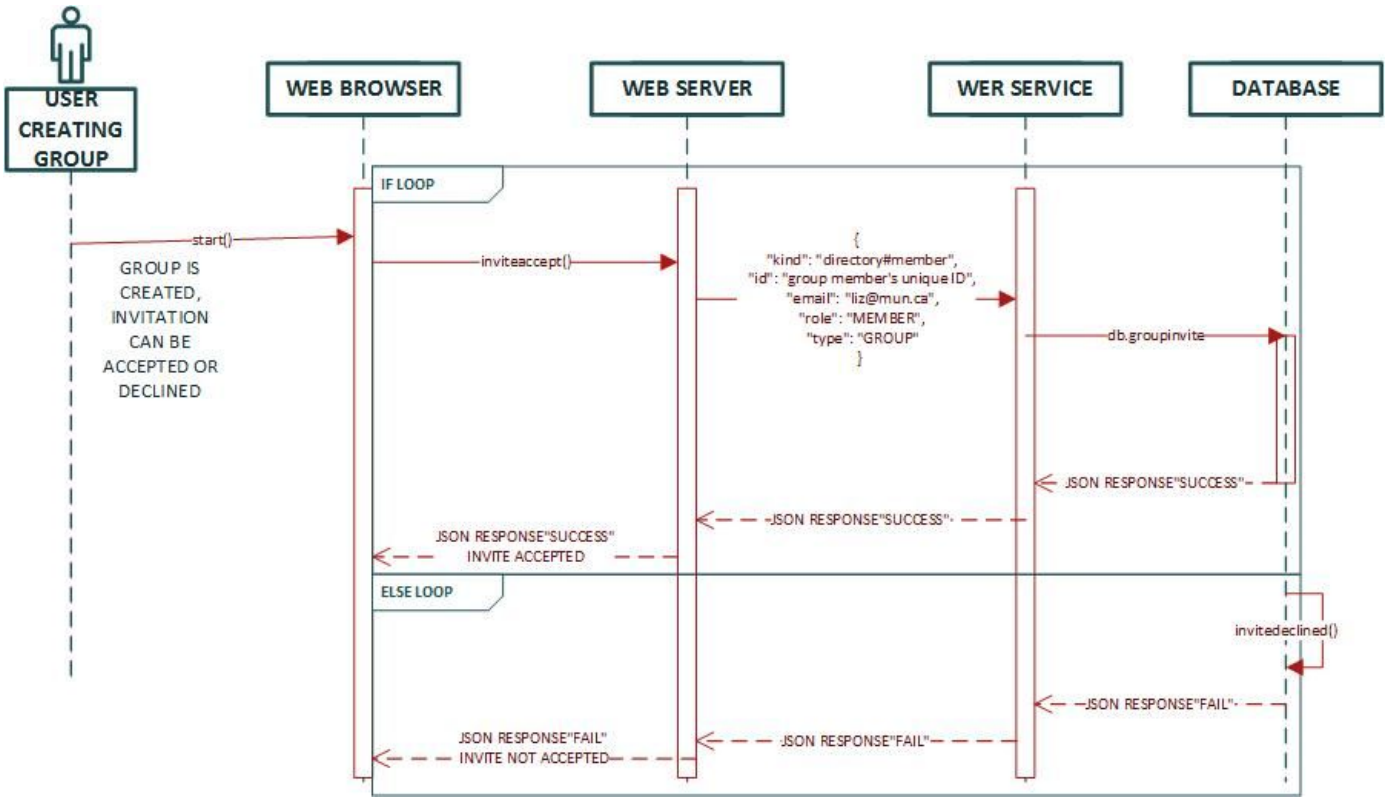


Figure SC-SG-03 - ACCEPTING STUDY GROUP INVITATION

SC-SG-04	**VIEW GROUPS**
Input	UserID of User
Sample URL	http://cs.mun.ca/s77ajc/MUNSN/userID/groups
Output	User will be able to view all the groups he's enrolled in
Sample JSON	{"groups" Group1, "groups" Group2}
Explanation	All groups will be shown in clickable links to open their group pages
Web-Server Request Type	GET

SC-SG-04	**VIEW A GROUP PAGE**

Input	UserID of User
Sample URL	http://cs.mun.ca/s77ajc/MUNSN/userID/groups/group01
Output	User will be able to view the group page he selects.
Sample JSON	{“groups” Group1}
Explanation	Any group a user is a part of can be clicked to be viewed to see activity thats going on in the group.
Web-Server Request Type	GET

	MODULE 3: RESUME
SC-RS-01	**Upload Resume**
Input	idUser, pdf file
Sample URL	http://cs.mun.ca/s77ajc/MUNSN/resume/userid
Output	Verification that upload was successful and a view of uploaded resume.
Sample JSON	{“success”:”<1 or 0>”
Explanation	The user is uploads a resume as a pdf which can then be viewed by others.
Web-Server Request Type	POST
SC-RS-02	**Delete Resume**
Input	userID
Sample URL	http://cs.mun.ca/s77ajc/MUNSN/resume/userid
Output	Verification that the existing resume has been deleted. Resume page no longer displays a resume.
Sample JSON	{“success”:”<1 or 0>”}
Explanation	User removes an existing resume
Web-Server Request Type	DELETE
SC-RS-03	**View Resume**

Input	userID
Sample URL	http://cs.mun.ca/s77ajc/MUNSN/resume/userid
Output	Webpage displaying user's resume
Sample JSON	{"Resume": [{ "resumeOwner": "userID" "resume_url":"<resume url>" }] }
Explanation	When user navigates to resume page, if they have an existing resume it will be displayed
Web-Server Request Type	GET

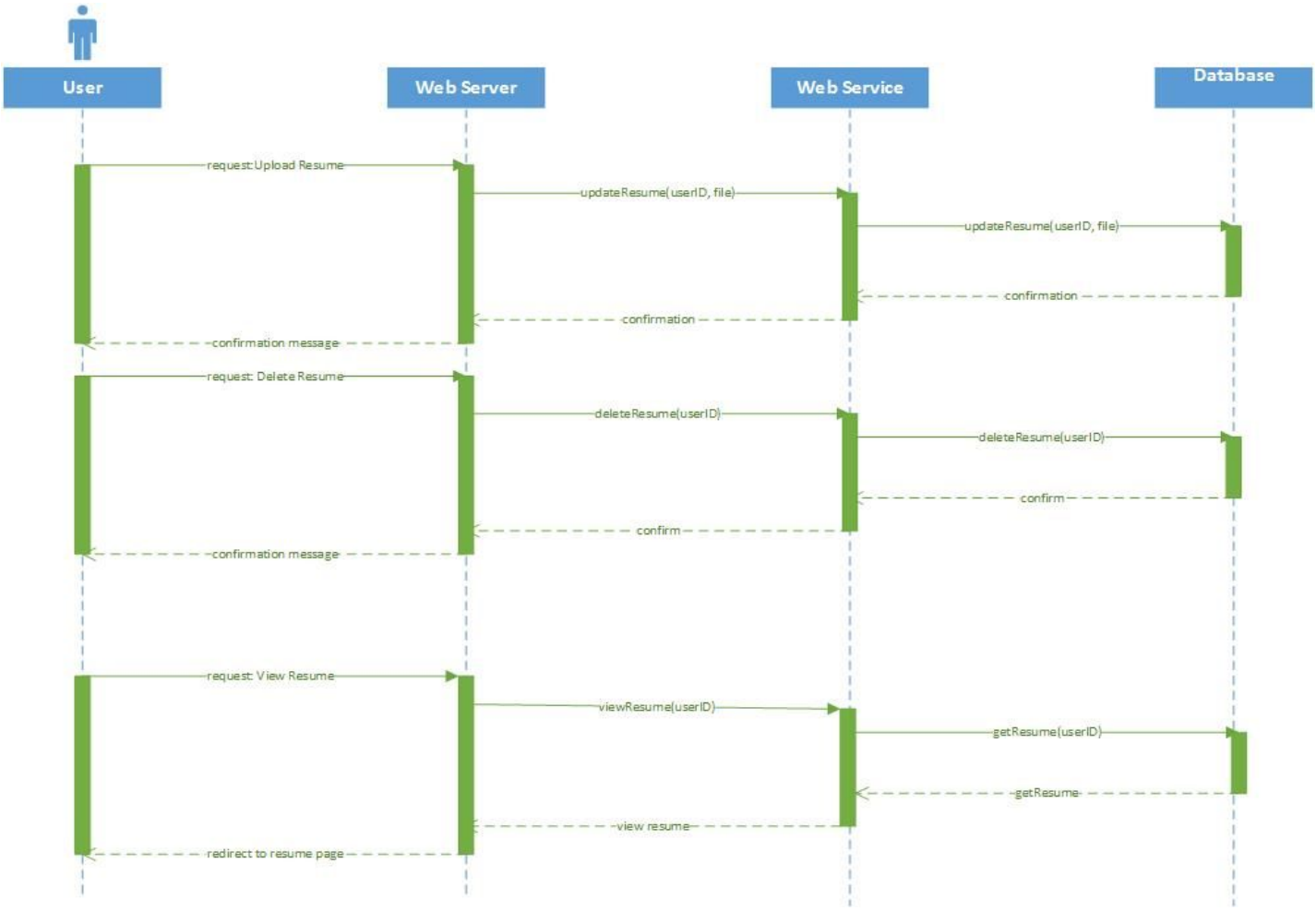


Figure SC-RS - UPLOAD/DELETE/VIEW RESUME

	MODULE 4: PRIVATE MESSAGING
SC-PM-01	**Read Messages**
Input	Parameters: current userID
Sample URL	http://cs.mun.ca/s77ajc/MUNSN/Messages
Output	Sets of text messages sent between two users. Grouped into conversations of unique participant pairs.
Sample JSON	{ “Conversation”: [{ “conversation_id”:”<conversation id>”,

	“participant_one”:”<user id>”, “participant_two”:”<user id>”, “Messages”: [{ “user_id”:”<user id of sender>”, “message_text”:”<message text>”, “date_time”:”<date time stamp>” }, { “user_id”:”<user id of sender>”, ... },] }, { “conversation_id”:”<conversation id>”, ... },] }
Explanation	Gets a list of all conversations involving the inputted user. For each unique pair of users that send a message to each other a conversation is returned along with all text messages for that conversation are given in chronological order.
Web-Server Request Type	GET
SC-PM-02	**Send Message**
Input	Parameters: sender userID, recipient userID Data: message text
Sample URL	http://cs.mun.ca/s77ajc/MUNSN/Message/Send
Output	Verification of successful creation of message
Sample JSON	{“success”:”<1 or 0>”}
Explanation	When a message is sent from one user to another a unique Conversation object is created for that user pair if there was not one already. The Conversation can contain any number of text messages each associated with the sending user and sorted chronologically.

Web-Server Request Type	POST

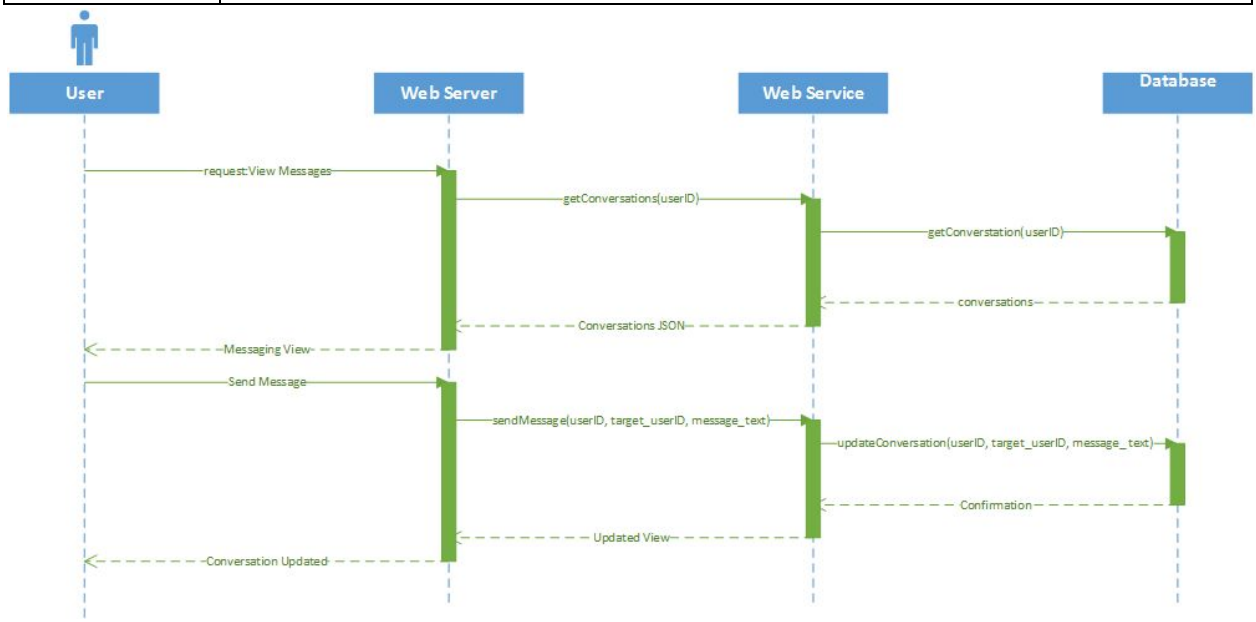


Figure SC-PM - VIEW/SEND

	MODULE 5: LOST AND FOUND
SC-LF-01	**CREATE LOST AND FOUND POST**
Input	UserID, Google API
Sample URL	http://cs.mun.ca/s77ajc/MUNSN/lostandfound/
Output	A marker on a map of the Lost and Found item.
Sample JSON	var map; // The JSON data var json = [{"id":48,"title":"Helgelandskysten","longitude":"12.63376","latitude":"66.02219"},{"id":46,"title":"Tysfjord","longitude":"16.50279","latitude":"68.03515"},{"id":44,"title":"Sledehunds-ekspedisjon","longitude":"7.53744","latitude":"60.08929"},{"id":43,"title":"Amundsens sydpolferd","longitude":"11.38411","latitude":"62.57481"},{"id":39,"title":"Vikingtokt","longitude":"6.96781","latitude":"60.96335"},{"id":6,"title":"Tungtvann-sabotasjen","longitude":"8.49139","latitude":"59.87111"}]; function initialize() { // Giving the map som options

	`` var mapOptions = { zoom: 4, center: new google.maps.LatLng(66.02219,12.63376) }; // Creating the map map = new google.maps.Map(document.getElementById('map-canvas'), mapOptions); // Looping through all the entries from the JSON data for(var i = 0; i < json.length; i++) { // Current object var obj = json[i]; // Adding a new marker for the object var marker = new google.maps.Marker({ position: new google.maps.LatLng(obj.latitude,obj.longitude), map: map, title: obj.title // this works, giving the marker a title with the correct title }); // Adding a new info window for the object var clicker = addClicker(marker, obj.title); } // end loop // Adding a new click event listener for the object function addClicker(marker, content) { google.maps.event.addListener(marker, 'click', function() { if (infowindow) {infowindow.close();} infowindow = new google.maps.InfoWindow({content: content}); infowindow.open(map, marker); }); } } // Initialize the map google.maps.event.addDomListener(window, 'load', initialize); ```
Explanation	A user can create a status to help a lost item be found with its location

	posted on a map and the item can be picked up from the user who created the post. The post is treated as a normal ‘create a post’ as done in the Timeline Module.
Web-Server Request Type	POST

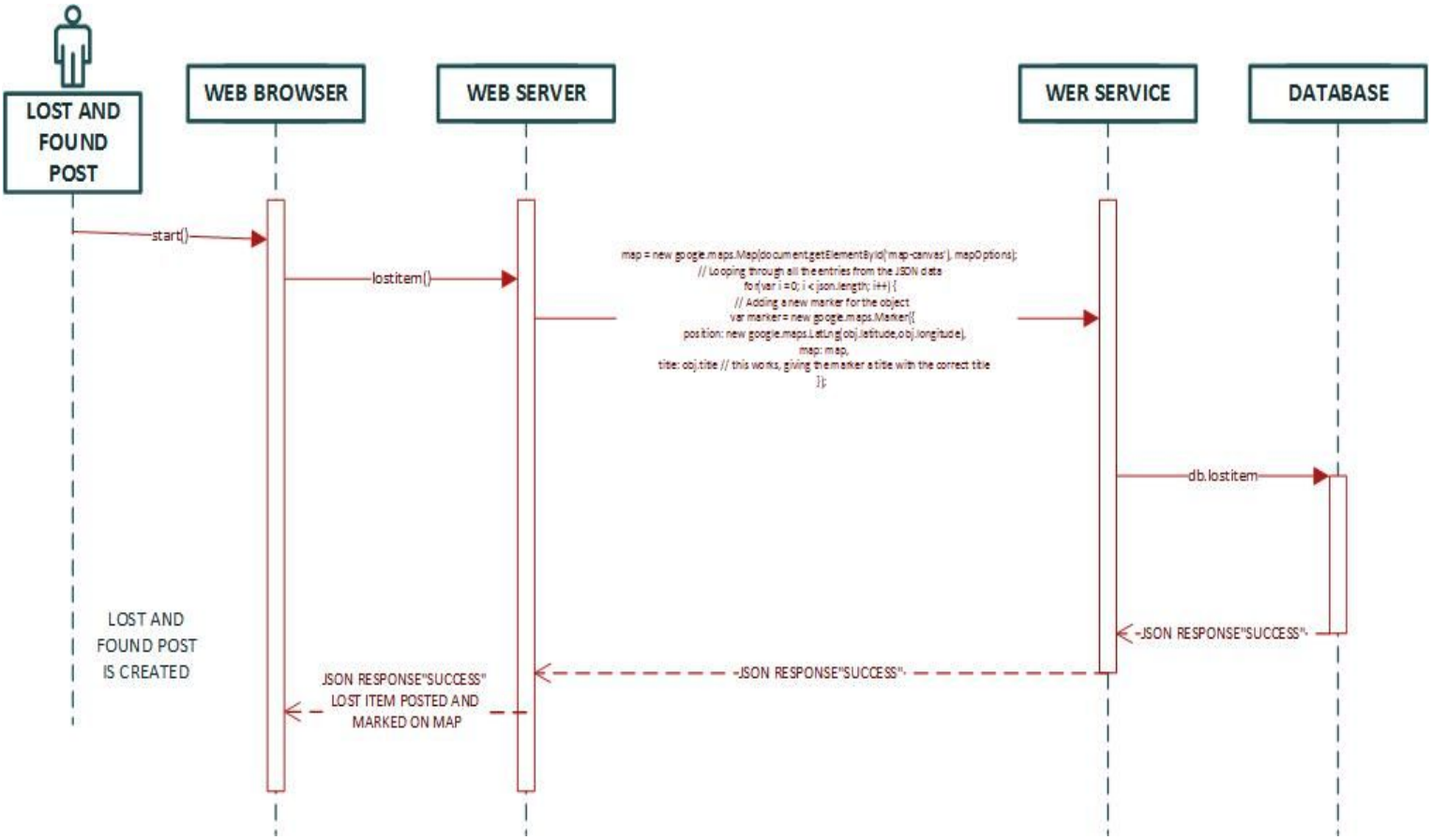


Figure SC-LF-01 - CREATE LOST AND FOUND POST

SC-LF-02	**RETRIEVING LOST ITEM**
Input	USERid
Sample URL	http://cs.mun.ca/s77ajc/MUNSN/lostandfound/
Output	User who lost the item will be directed to the Posters page and then provided a phone number so then the user may contact the poster.
Sample JSON	

Explanation	A lost item can be retrieved by a user contacting the poster through a cell phone number or by private messaging where they will negotiate on how the lost item is going to be retrieved.
Web-Server Request Type	POST

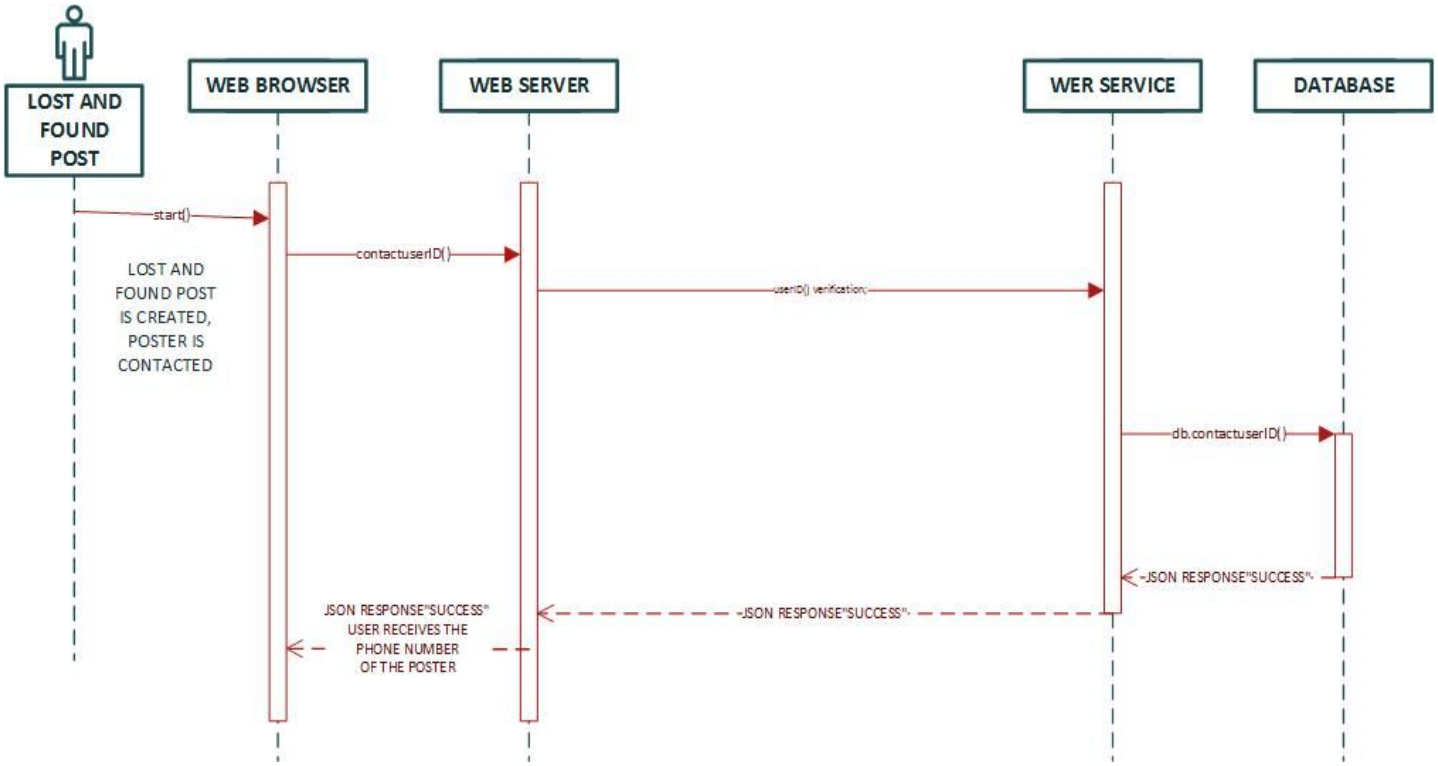


Figure SC-LF-02 - RETRIEVING LOST ITEM

	MODULE 6: POLLS
SC-PO-01	**Create Poll**
Input	userID, classID(for specific class), pollContent(name of poll, options to vote on)
Sample URL	http://cs.mun.ca/s77ajc/MUNSN/timeline/view?timeline_id=1234567&user_id=7654321
Output	Content; a poll viewable only by those who are authorized (have same course and are friends of poll creator)
Sample JSON	{“Poll”: [

	{“poll_ID”:”<poll identification number>”, “pollOwner”:”userID”, var poll = { “topic” : “<title of poll>”, “option1”:”<option1>”, “option2”:”<option2>”, “option3”:”<option3>”, “option4”:”<option4>”, “numVote1”:”0”, “numVote2”:“0”, “numVote3”:”0”, ”numvote4”:”0”};] }
Explanation	User creates a poll with a specific course as the topic and x number of options.
Web-Server Request Type	POST
SC-PO-02	**Vote on Poll**
Input	userID, pollVote
Sample URL	http://cs.mun.ca/s77ajc/MUNSN/timeline/view?timeline_id=1234567&user_id=7654321
Output	Update to the poll results
Sample JSON	{“Poll”: [{“poll_ID”:”<poll identification number>”, “pollOwner”:”userID”, var poll = { “topic” : “<title of poll>”, “option1”:”<option1>”, “option2”:”<option2>”, “option3”:”<option3>”, “option4”:”<option4>”, “numVote1”:”<Number of votes>”, “numVote2”:“<Number of votes>”, “numVote3”:”<Number of votes>”, ”numvote4”:”<Number of votes>”};] }
Explanation	A user votes on one of the options (pollVote) which is sent to the DB to update the results of the poll. The userID is also stored to prevent that user from voting again.
Web-Server Request Type	POST

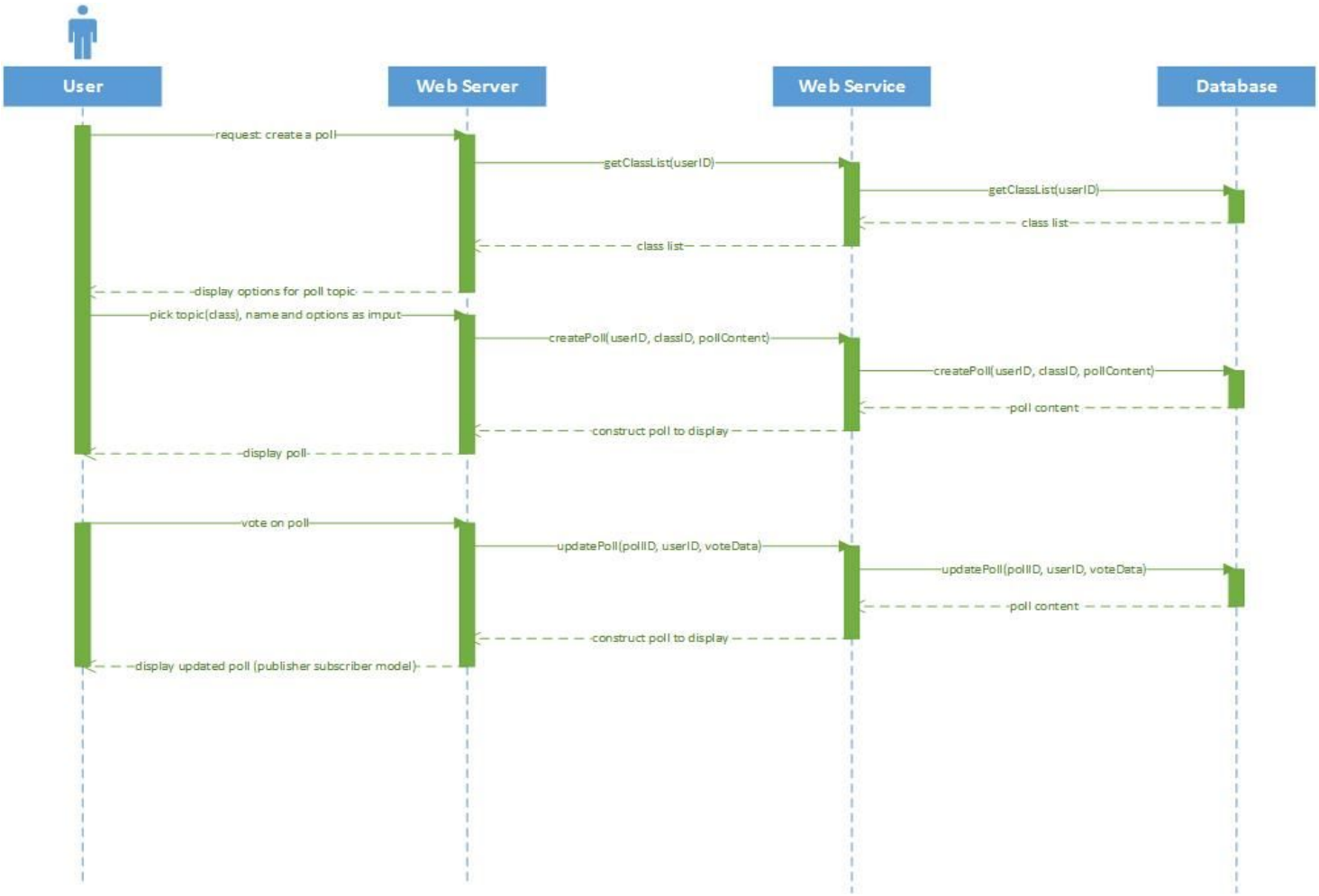


Figure SC-PO - CREATE/VOTE POLL

	MODULE 7: FRIENDS
SC-FR-01	**View Friend Recommendations**
Input	Parameters: userID
Sample URL	http://cs.mun.ca/s77ajc/MUNSN/Friends/Recomendations
Output	A sorted list of recommended friends customized for the current user. For each recommendation a name, userID and profile picture are given.
Sample JSON	{“Recommendations”: [{ “user_id”:”<recommended friend user id>”, “username”:”<recommended friend username>”,

	“profile_picture_url”:”<profile picture url>”, }, { “user_id”:”<next recommended friend user id>”, ... }] }
Explanation	Upon request the software creates a list of recommended Friends for the current User. The list is based off weighted factors such as number of Friends in common and shared Study Groups. The list is returned in descending order from most connected to lowest.
Web-Server Request Type	GET
SC-FR-02	**Send Friend Request**
Input	Parameters: current userID, target userID
Sample URL	http://cs.mun.ca/s77ajc/MUNSN/Friends/Request
Output	A Friend connection is created between the current User and the target User.
Sample JSON	
Explanation	Upon sending a Friend Request an object is created connect both Users. The object has a state variable that is initially set to “unaccepted”. If both Users send a request to the other than the state is changed to “accepted” and the Users will appear on each other's Friends lists.
Web-Server Request Type	POST

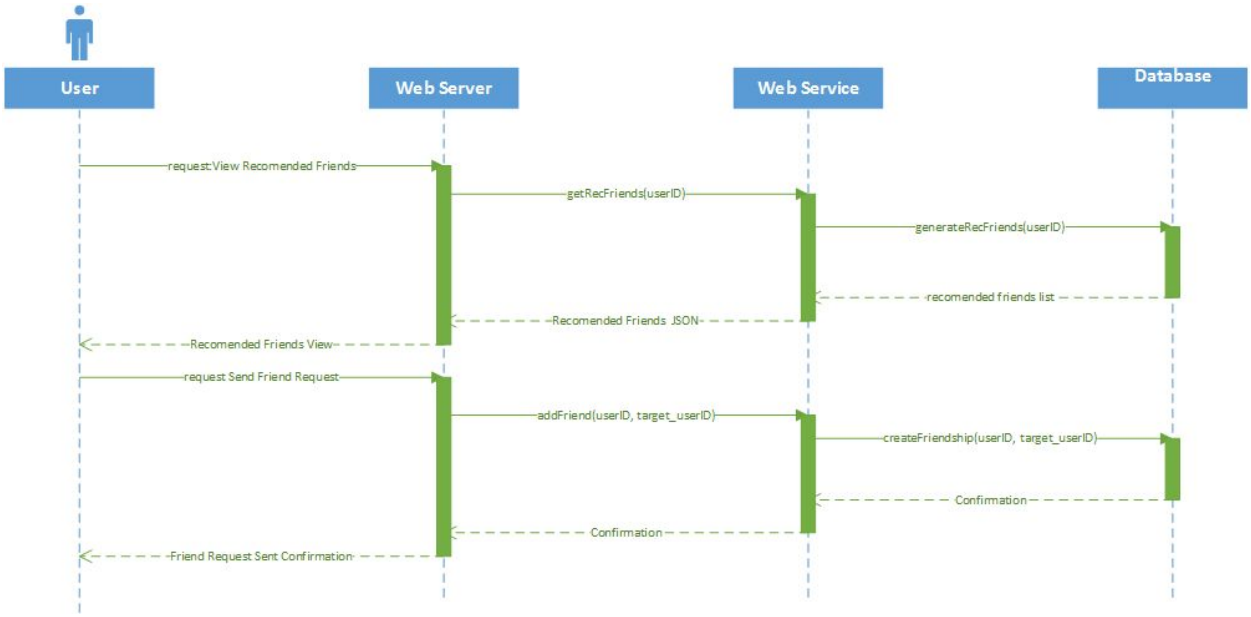


Figure SC-FR-01-02 - VIEW RECOMENDED/SEND REQUEST

	MODULE 7: FRIENDS
SC-FR-03	**View Friend List**
Input	Parameters: userID
Sample URL	http://cs.mun.ca/s77ajc/MUNSN/Friends
Output	A list of Friends of the current user sorted alphabetically.
Sample JSON	{“Friends”: [{ “user_id”:”<Friend user id>”, “username”:”<Friend username>”, “profile_picture_url”:”<profile picture url>”, }, { “user_id”:”<next friend user id>”, ... }] }
Explanation	A list of all Friends of the current User is returned in alphabetical order. For each Friend a profile picture, username and userID are returned.

Web-Server Request Type	GET

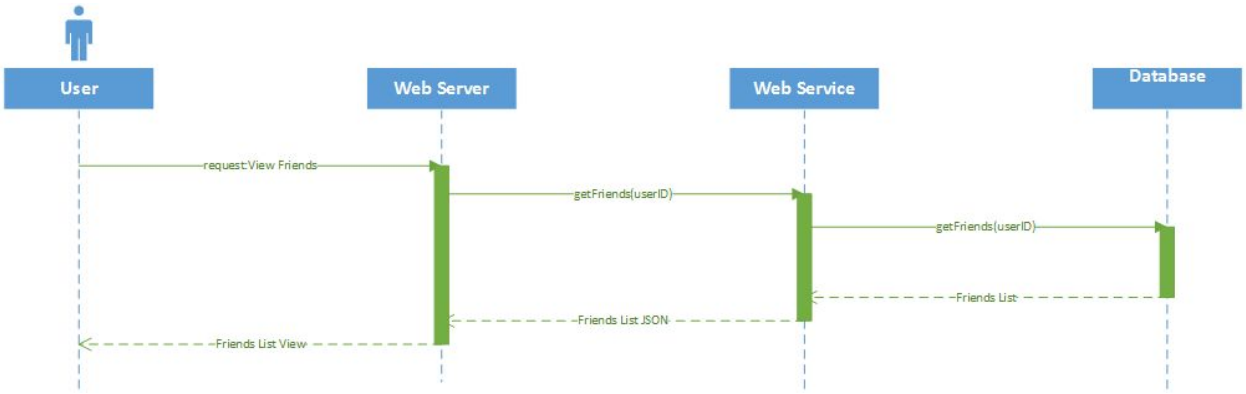


Figure SC-FR-03 - VIEW FRIENDS LIST

	MODULE 8: USER ACCOUNT
SC-UA-01	**Create Account**
Input	Parameters: username, password, email
Sample URL	http://cs.mun.ca/s77ajc/MUNSN/Account/Create
Output	Returns a success or failure. If success also sends a confirmation email to the given address.
Sample JSON	
Explanation	The software confirms that each of the parameters meets their respective criteria e.g., the email must be unique and end in “@mun.ca”. If the confirmation succeeds a confirmation email is sent to the given email. If the confirmation fails a notification of failure is returned.
Web-Server Request Type	POST
SC-UA-02	**Confirm Account Email**
Input	Parameters: confirmation code
Sample URL	http://cs.mun.ca/s77ajc/MUNSN/Account/Confirm/avd142as7dg235
Output	A success or failure notification depending on the validity of the code.
Sample JSON	
Explanation	For email confirmation the user is given a code and a unique url to visit in

	order to submit the code. If the code entered on the page is the correct code for the url then the related account is confirmed valid and is unlocked for use.
Web-Server Request Type	POST

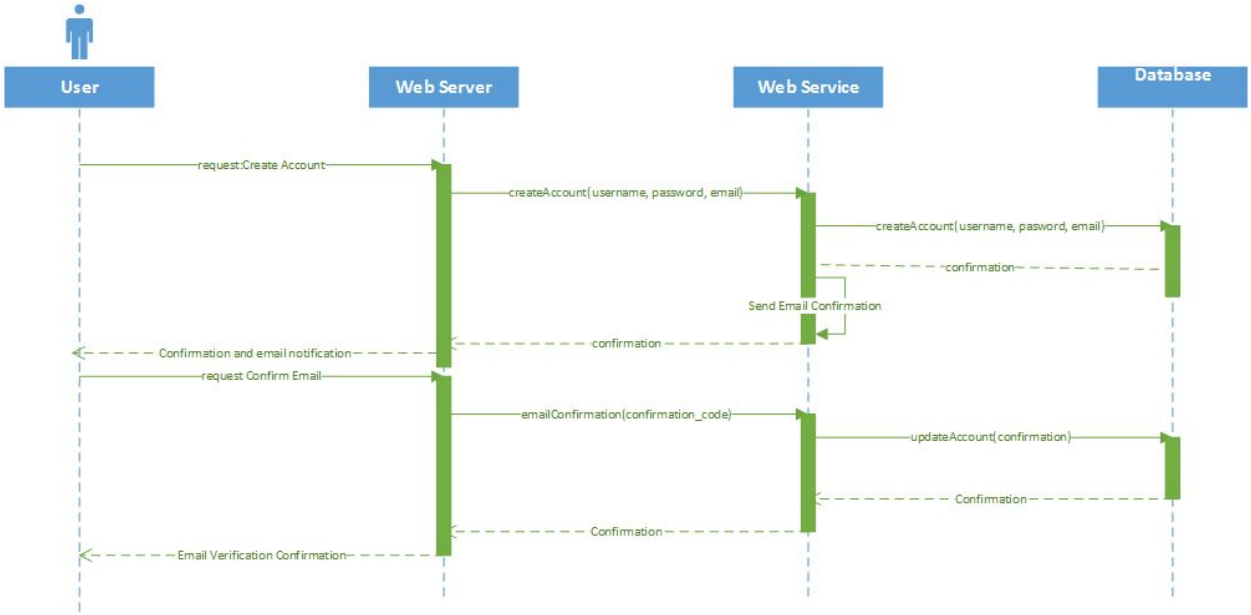


Figure SC-UA - CREATE/CONFIRM

	MODULE 9: SCHEDULE
SC-SC-01	**Add to Schedule**
Input	userID, classID
Sample URL	http://cs.mun.ca/s77ajc/MUNSN/schedule/
Output	An updated version of the user's schedule.
Sample JSON	{“success”:”<1 or 0>”}
Explanation	User updates their existing schedule. A schedule that is empty still exists.
Web-Server Request Type	POST
SC-SC-03	**Remove from Schedule**
Input	userID, classID
Sample URL	http://cs.mun.ca/s77ajc/MUNSN/schedule/

Output	An updated schedule with the removed item no longer present and verification to the user that the item was removed successfully.
Sample JSON	
Explanation	Remove a class from the schedule.
Web-Server Request Type	POST
SC-SC-03	**View Schedule**
Input	userID
Sample URL	http://cs.mun.ca/s77ajc/MUNSN/schedule
Output	The current schedule of the user
Sample JSON	{“schedule”: [“classList”: { “class1”:”<class 1>”, “class2”:”<class 2>”, “class3”:”<class 3>”, ... }] }
Explanation	USer navigates to schedule page this will be the default call, they will be presented with their current schedule.
Web-Server Request Type	GET

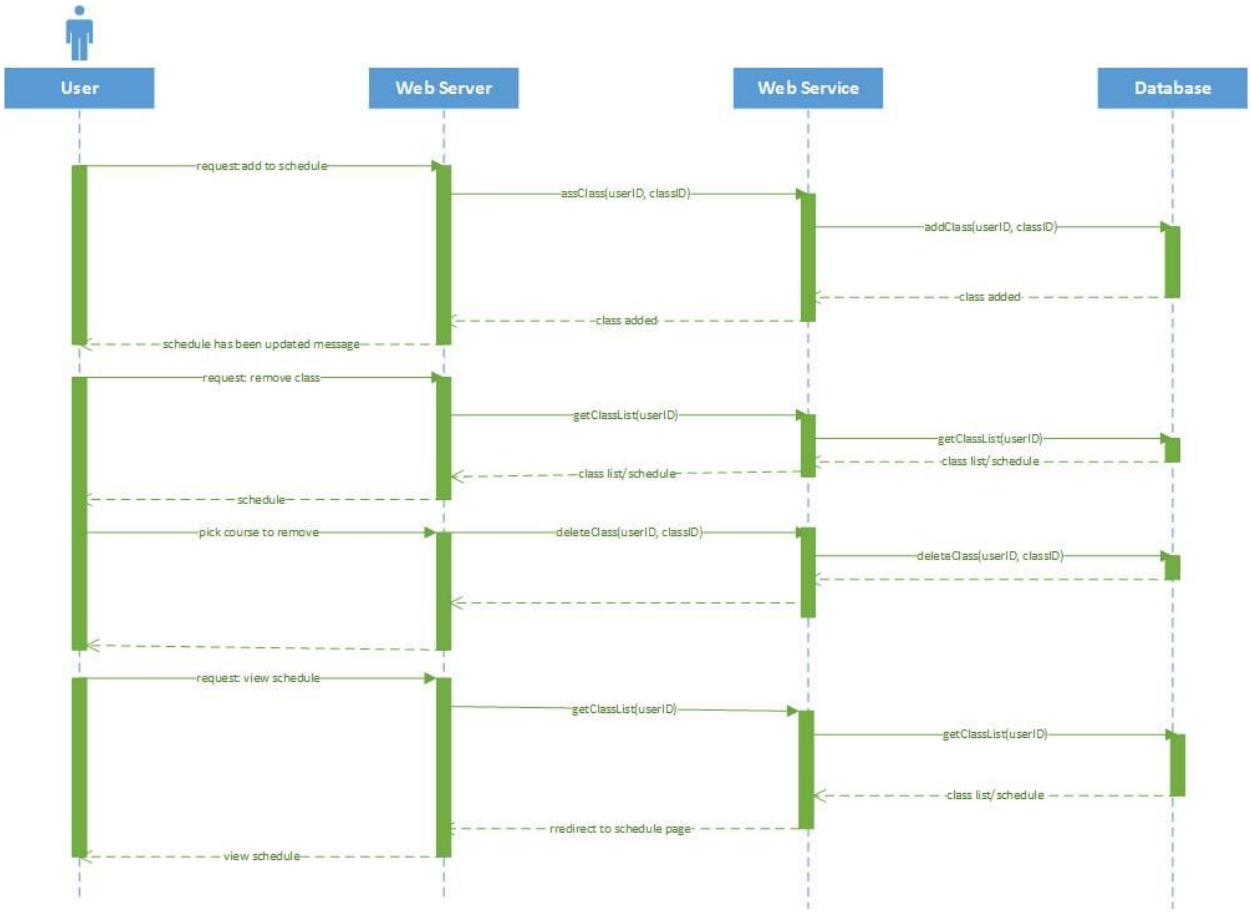


Figure SC-SC - ADD/REMOVE/VIEW SCHEDULE

MongoDB Logical Model

The mongoDB schema is built according to access patterns. This is why we listed all the service calls to the web service. The table below shows which mongoDB collections are involved for each service call.

Service Call	**Input**	**MongoDB Collections**
TIMELINE		
View a timeline	Timelines user_id	Posts, timelineRights
Create a Post	Timeline user_id, poster user_id, datetime, post text,	Posts, timelineRights

Comment on a Post	Post_id, commenter user_id, datetime, comment text	Posts, timelineRights
Edit a Comment	Post_id, comment_id, editor user_id, datetime	Posts, timelineRights
View comment edit history	Post_id, comment_id, user_id of viewer	Posts, timelineRights
Reply to a comment	Post_id, comment_id, user_id of replier, comment reply text	Posts, timelineRights
Change Timeline Posting Rights	User_id, posting right type	timelineRights
Change timeline Visibility Rights	User_id, timeline visibility right type	timelineRights
Change Timeline individual post visibility	User_id, post_id, visibility type	posts
STUDY GROUP		
Create Study Group	User_id, GroupRight Type	Groups, GroupRights
Send Study Group Invite	Group_id, user_id of user who will receive request	Groups, GroupRights
Accept Study Group Invite	User_id, group_id	Groups, GroupRights
View Study Groups	user_id	Groups, GroupRights
View Individual Study Group	User_id, group_id	Groups, GroupRights
RESUME		
View resume	User_id of resume page owner	Resume
Edit Resume	user_id	Resume
Delete Resume	user_id	Resume
PRIVATE MESSAGING		
Read message	User_id, chat_id	n/a
Send message	User_id, chat_id	n/a

LOST AND FOUND		
Create Lost and Found Post	User_id, image, coordinates, datetime	LostandFound
View Lost and Found Posts	None required	LostandFound
Reply to lost and Found Post	user_id	LostAndFound, Friends
Contact Poster	user_id	LostAndFound, Friends
POLLS		
Create a poll	user_id	Polls, Users
View a Polls	user_id	Polls
Vote on a Poll	User_id, poll_id	Polls
FRIENDS		
View Friends List	user_id	Friends
Send Friend Request	User_id, proposed_friend_user_id	Friends
Accept Friend Request	User_id, friend_user_id	Friends
View Suggested Friends List	user_id	Friends
ACCOUNT		
Create Account	User_id, email	Users
Confirm email account	user_id	Users
SCHEDULE		
View Schedule	user_id	Schedule
Add Class to Schedule	User_id, class, timeslot	Schedule
Remove class from Schedule	User_id, class_id	Schedule

MongoDB Collections

The following mongoDB collections are required according to the list of service calls.

1. Users
2. Posts - Comments are an embedded collection as comment history.
3. Timeline Rights

4. Groups
5. GroupRights
6. Resume
7. Friends
8. LostAndFound
9. Schedule

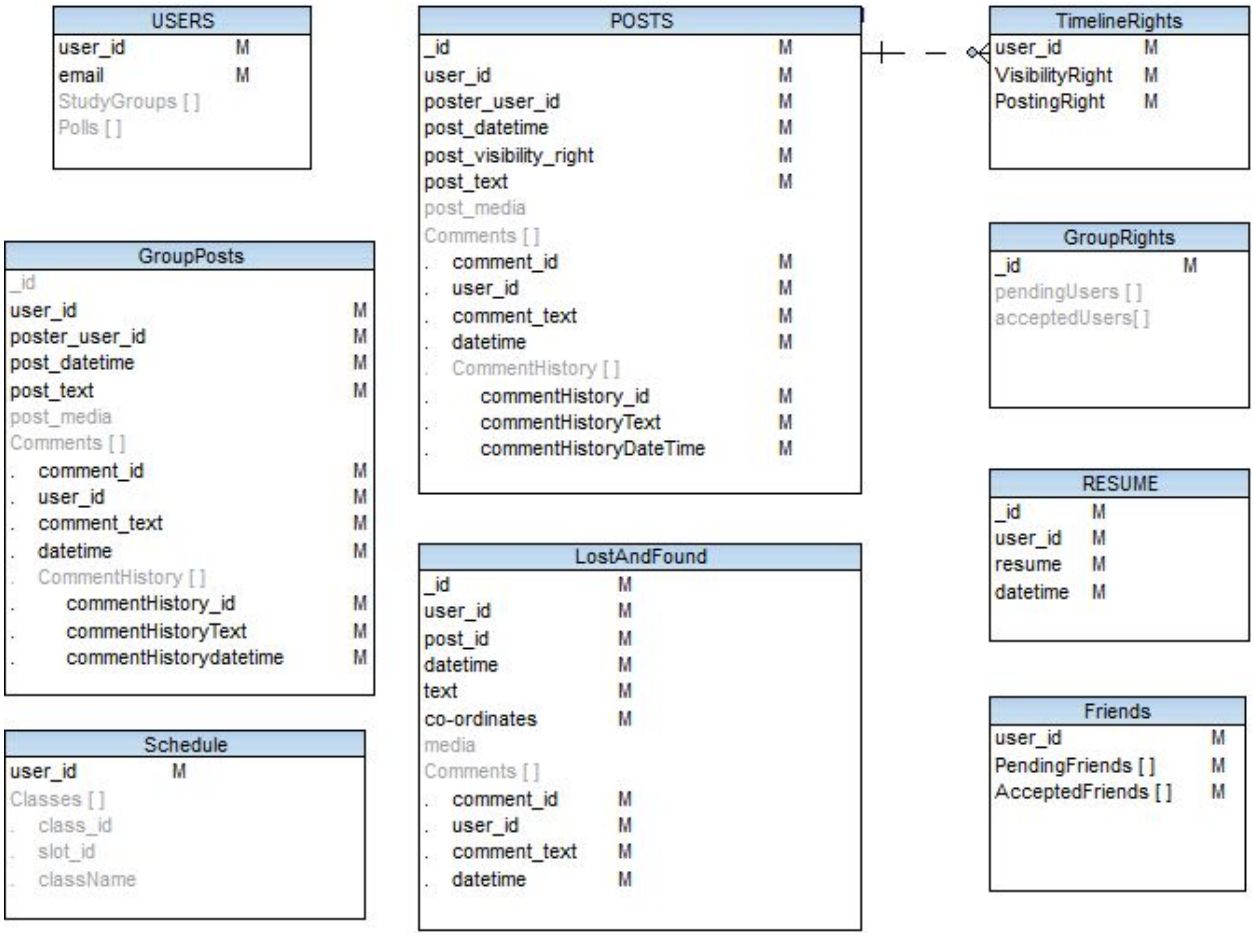


Figure - Mongodb Logical Diagram

Responsibilities

Document Responsibilities

- Saahil - Study Group Module, Lost and Found Module, Class Model Diagram
- Mark - Polls Module, Resume Module, Schedule Module
- Allan - Timeline Module, Mongodb Logical Model, Overview, Document Structure
- Tyler - Account Module, Private Messaging Module, Friends Module

Module Responsibilities

In keeping with an agile methodology we are only assigning responsibility for one module at a time. When that module has passed unit / integration and regression testing then the developer can get another module to work on. This way we are not locking up a block of work that another can be working on.

Currently assigned modules are as follows;

- Timeline Module - Allan - all notifications will use this
- Account Module - Tyler - we must be logged in to do any testing
- Study Group Module - Saahil
- Schedule Module - Mark - the study group module requires the data that schedule module will push to the mongodb back-end.

The plan for work going forward is that we will get the back-end set up and communicating with the front-end. The back-end lead is Allan and the front-end lead is Tyler. The back-end team will ensure that,

1. the mongodb database is set up
2. 9 collections are set up
3. Web server is set up
4. Web Service is set up
5. All components communicate with each other.

The front-end team will ensure that,

1. The HTML skeleton is set up using bootstrap
2. Skeleton is mobile friendly and responsive
3. WebCommunications module is set up and handles GET and POST requests and other functions can use it to send and recieve data.
4. The main model, main view, main controller are set up and adding the additional mvc models will be easy.

Once we have the front-end and back-end components in place and communicating we can begin work on the front-end angularjs mvc models.

Scheduling

HTML skeleton, Client-Side MVC models, Web Server, and Web Service can all be developed independently.

Week 1(5-1-17 to 13-1-17):

- **Tyler Vey**: Use Cases, Functional and Nonfunctional Requirements from project description 1-3.

- **Saahil Budhrani**:Use Cases, Functional and Nonfunctional Requirements from project description 7-9.
- **Mark Hewitt**:Use Cases, Functional and Nonfunctional Requirements from project description 9-12.
- **Allan Collins**:Use Cases, Functional and Nonfunctional Requirements from project description 4-6.

Week 2(14-1-17 to 21-1-17):

- **Tyler Vey**: General improvements on SRS Formatting and distribution of sections 1, 2 and 4.
- **Saahil Budhrani**: General improvements on SRS Formatting and distribution of sections. Worked on sections 3.5(Use Case diagram), 3.6, 3.7, 4 and powerpoint presentation.
- **Mark Hewitt**: General improvements on SRS Formatting and distribution of sections. Worked on definitions and section 3.0 through 3.5
- **Allan Collins**: General improvements on SRS Formatting and distribution of sections. Worked on sections 1.2, 2.3, 2.4, 3.2 - 3.4, 3.5.4, 3.5.5, 3.5.6, 3.5.7, 3.6.

Week 3(22-1-17 to 28-1-17):

- Finalization of SRS document as a group and minor formatting.

Week 4(29-1-17 to 5-2-17):

- Powerpoint presentation
- Milestone 1 presentation.
- Work commences on Architectural Document, system's decomposition into modules.

Week 5(6-1-17 to 12-2-17):

- Division of module responsibility between team members.
- Creation of UML and other visualisations.
- Progress on Architectural Document.

Week 6(13-1-17 to 19-2-17):

- Finalization of Architectural Document.
- Work commences on Module Documents.

Week 7(13-1-17 to 19-2-17):

- Work on descriptions of the functionality and interface of each module.
- Work on descriptions of module testing plans and demonstration.
- Back-end / Front-end teams commence work on back-end / commincations

Week 8(20-1-17 to 26-2-17):

- Continued work on descriptions of the functionality and interface of each module.
- Continued work on descriptions of module testing plans and demonstration.
- Back-end is complete and front-end module work commences.

Week 9(27-1-17 to 5-3-17):

- Finalization of Module Documents.
- Preliminary System Development.

Week 10(6-17 to 12-3-17):

- Creation of Modules based on Module Documents.

Week 11(13-17 to 19-3-17):

- Integration of all modules.
- Creation of System Testing Document.

Week 12(20-3-17 to 25-3-17):

- Deployment of Integrated System.
- Creation of Data for System Demonstration and testing.
- Improvement upon System Testing Document.

Week 13(26-17 to 2-4-17):

- Completion of Integrated System.
- Submission of System Testing Document.
- Milestone 4 presentation.