TEAM DOCUMENTATION

DormUtils

Web Application

CSCI 201 Group 4

Team Members: Silas John, Qilin Ye, Andy Gu, Arya Wang, Gabe, Ben, Jacob Ma

Table of Contents

CSCI 201 Project Proposal - Group 4

Overview:

We are developing a web application tool which manages shared expenses and financial responsibilities between roommates. Our web application will have support for both guest and authenticated users. Users can be placed in "cohorts," a virtual representation of real-life living "groups." Each cohort features a shared shopping list containing supplies and necessities for the group.

Features:

(1) Shopping List:

- Each member of the cohort can add desired items to the shopping list
- Before each purchase is made, members can still remove unwanted items
- Multi-Threaded (real-time) sort of items displayed every time a new list or item is added
- Multiple users can add and remove items to the shopping list at the same time (multithreading for sending and receiving requests from multiple users to one server)
- Implements an internal algorithm to assign specific users within cohorts a unique list of items to purchase, standardizing the total expense by user over time.

(2) Cohorts:

- Only authenticated users can join a cohort
- Each cohort features a personalized dashboard with information on your next purchases, total money spent, your cohort mates, etc.

High Level Requirements - CSCI 201 Group 4

General Overview

We have been requested to design and create a web application which compiles and catalogs living expenses. The web application is accessible to three intended audiences: guest users, authorized individual users, and authorized cohort users. A cohort is defined as a group of users who have reason to share expenses (a common example is college roommates sharing living expenses for food and basic necessities). All users can add items to a global, synchronized list of desired "shared" necessities and products and their corresponding cost. Each cohort shares a single list while guests and individual users (intrinsically considered to be "cohorts of one") will each have their own lists. Guest sessions are not "saved" sessions. Any authorized user's session should be "saved". The web application should split total costs approximately equally among each cohort member. An internal algorithm should "assign" specific itemized expenses to specific cohort members to accomplish this purpose. This application can store all information for authorized users and establish cohorts for the next time someone logs in. Multiple users in a cohort should be able to modify and view the list simultaneously.

We have also been requested to split up behavioral components of the web application into distinct webpages. Each web page should have a navigational feature to access other web pages. The webpage details are as follow:

First Webpage ~> Login Web Page

- *Guests should have an option to continue to the website without logging in (they have limited access)*
- *REGISTER: Users of the website should be able to create an account on this webpage*
- *LOGIN: Authorized users with an account should be able to login to the website on this webpage*

Second Webpage ~> Home Page

- *The web application should include an aesthetically pleasing dashboard / user interface tailored for each authorized user*
- *The web application should show relevant data for each authorized user (i.e. cohort information, miniature view of the list from webpage 4, items assigned to the user for purchase, etc.). Users should be able to mark items as purchased on this webpage.*
- *The dashboard should contain relevant statistics (i.e. purchasing costs to date)*

Third Webpage ~> Account Management Page

- *This webpage should contain necessary features for account and cohort management. Some required features and behaviors are listed below:*
- *CREATE: Authorized users should be able to create a uniquely identifiable cohort on this webpage*
- *JOIN: Authorized users should be able to join a cohort on this webpage (users are limited to being a part of one cohort per account)*

- *LEAVE: Authorized users should be able to leave the cohort they are currently part of*

<u>Fourth Webpage ~> Expense List Display Page</u>

- *This webpage contains a global, synchronized list of desired "shared" necessities and products and their corresponding costs and quantities.*
- *At a minimum, the following actions should be near-instantaneous for all users…*
    - *ADD: Authorized Users should be able to add items to this list*
    - *DELETE: Authorized Users should be able to remove items from the list*
    - *VIEW: Authorized users should be able to view both the list as well as any updates from other authorized users (if they are in the same cohort). They should also be able to view items assigned to themself.*
- *Items should be automatically assigned to members of a cohort in a manner that divides the total purchasing expenses to date as evenly as possible between each cohort member. Users are responsible for purchasing the items assigned to them.*

Group Names: Silas John, Qilin Ye, Gabriel Do, Benjamin Wiencko, Arya Wang, Jacob Ma

Technical Specifications

Login page - 20 hours

This webpage provides a staging area for users to login to the website. There are three different variations of the login page: one for those creating an account, one for those logging in with an authorized account, and one for joining as a guest. There are options on each page to reach the other variations.

- The login page contains a login component and an aesthetically pleasing UI design. This will consist of five main components.
    - The *main login component* includes a username field and a password field on the right side of the screen, with a login button below which symbolizes a "submit" function.
    - The *register component* includes a username field, a password field, and re-enter password field (to ensure the password is entered correctly). A register button will be below.
    - A *"Continue as Guest"* button with a different color enables users to continue as unauthorized users.
    - A *hamburger menu* will appear, navigating to other pages after clicking signing in. The authorized user will see "Welcome, {Username}!" on the top of the nav bar, log out at the bottom, and hyperlink to the dashboard and shopping list. Guests have a different color theme for the nav bar and also for all the backgrounds for all the pages, the hyperlink for the account management page is disabled for guests.
- The front end includes input validations and verifications. Error messages will be displayed based on the situation, including not giving an email address, the two inputted passwords not matching, or there existing an account under the email address already.
- At the backend, we will have a database storing and validating the user's ID and passwords. Registering an account will add data pairs to the backend database, and logging in requires validation from the database.
- A networking protocol will be needed to exchange data between the back-end database and front-end login components.

Dashboard UI / Home page - 25 hours

This webpage provides an overview of the other webpages. It is considered to be the "home" web page and will be the first page a user sees when they log in. This dashboard will be specialized and unique to each and every user.

~> The dashboard should display a unique welcome message to the user.

~> The dashboard should be visually pleasing, displaying information in an easy-to-read format.

~> This webpage should have a navigation bar (consistent upon every web page) for navigating to other pages.

~> This dashboard should have four other main components:

- Miniature View of the Sync List - The dashboard should contain a miniaturized view of the list. It will utilize some criteria to sort which items and which bits of information are displayed (aka sorted by most recently added items).
- Statistics - The dashboard should depict some relevant statistics for the user. Statistics include the amount of money needed for the whole cohort and the individual user. Statistics should be displayed with a visual graphing aid (aka a pie chart depicting the money spent as compared to other members in the cohort). Guests would need to sign in to unlock this function.
- Cohort Information - If the user is a part of the cohort, then the dashboard should depict the name of the cohort as well as the first few members of the cohort.
- Assigned Items - The dashboard will also display (if any exist) items assigned to the user by an internal algorithm in the web application. There will be buttons to check off assigned items indicating that a user has purchased the respective items. This is the only change that the user can actively activate for the backend. Guests would need to sign in to mark items as purchased.
- Shopping List - The dashboard will also display (if any exist) items in the entire user's cohort on the right side of the page.

~> Additionally, all of the information unique to the user in the dashboard / UI will be stored in a backend database. Any updates to the information via the dashboard (aka marking off priority items) will first update the database in the backend, then the backend will correspondingly update the dashboard UI.

~> The dashboard will utilize both the internal algorithm for assigning items to specific users as well as additional algorithms to create unique statistics for each user.

Synchronized List Webpage - 25 hours

This webpage displays and handles the logic behind the synchronized list shared between all members of a cohort. Every member of a cohort should see an identical page. The page handles both add requests for an item, and the actual printing of the list.

~> Shopping List Web Page - Adding an Item Section

- The web interface needs to have several fields where a user can input all the information about an item they wish to add to the list
    - These fields will be located near the top of the page, above the list
- There should be a field for the item name, quantity, and price
- On the left side of these fields should be an "add to list button."
    - If the user presses this button and one of the fields is empty, the webpage should display an error message.
    - If all the fields are filled, a form should be updated and sent to the backend.

~> Shopping List Web Page - List Display Section

- Each element of the list has an item name, quantity, price, and assigned group member (whoever was assigned to purchase the item)

- After an element has been added to or removed from the list, the list should immediately update to reflect the change
- If another member of a cohort edits the list, all other members should be able to see the change after refreshing the page

~> List Backend and List Database

- Each element of the list will be represented with an Item class
- Item has all of the data pertaining to a single item in the shopping list, including entry_id, entry_name, entry_quantity, entry_individual_price, etc.
- The Cohort class has a list of Items as a member variable
    - The Cohort class also has a list of Users as a member variable.
- The database has a table whose each row corresponds to a cohortID + ShoppingItem.
    - This table stores every item ever added to any shopping list
- The server can fetch entries in the table based on the cohortID and place them in a Item list when needed
    - This minimizes the number of times the database is accessed
- When processing, adding and remove requests, the server should use multithreading to handle simultaneous requests
    - Each Cohort will be a synchronized resource to prevent race conditions.
    - This should fulfill the networking requirement, as multiple clients can request from the same server

~> Assigning List Items to a User

- When a user is assigned an Item to buy, the price of the Item is added to that user's proj_money_spent (projected money spent) variable.
- When an item is purchased, the price of the Item is added to that user's money_spent variable.
- When assigning an item to a user to buy, the proj_money_spent variable for every member of the cohort will be checked, and the user who has spent the least amount of money will be assigned the newest item.

Account Management WebPage - 15 hours

This webpage interacts heavily with the backend. You can manage your own account here as well as create and join and leave cohorts (creating and deleting new data in the backend).

- The frontend web page will have an aesthetically pleasing UI and three major account management components (likely positioned side-by-side):
- If the user is not in any cohort, the page has the following functionality:
    - *CREATE*: Authorized users should be able to create a uniquely identifiable cohort on this webpage. A create section consists of a cohort name field, a cohort description field (optional input), and a create button. Successfully creating a cohort will assign a user with a unique cohort ID and directly direct to the Home Page. Unsuccessful cohort creation will lead to on-screen alerts and help.

    - *JOIN*: Authorized users should be able to join a cohort on this webpage (users are limited to being a part of one cohort per account) A join section consists of only a cohort ID input field (where you enter a unique ID predetermined by the creation of another cohort) and a join button. Clicking on the "Join" button will make the user successfully join the cohort and refreshes the content on the current age. The above will only activate if a valid ID is entered; if not, an error message will display.
- If the user is in some cohort, the page has the following functionality:
    - *LEAVE*: Authorized users should be able to leave the cohort they are currently part of. With a red box printing "Leave Current Cohort" on the button of the page. Upon leaving a cohort, items assigned to a user will be released from the assignment and reassigned to other users in the cohort.
    - Display up to 10 members in the same cohort in two lines, displaying each name in a nice-looking bubble\tag.
- A certain algorithm will generate a unique cohort ID (identifier) based on the information inputted and store it in the backend database pointing to the relevant cohort data. This identifier will also be able to extract certain cohort information from the database.
- A backend object storing cohort IDs and corresponding cohort information needs to be created and updated with each of the three above functions.
- A networking protocol will be needed to exchange data between the back-end database and front-end login components.
- *Guests do not have this page.*

Webpage Agnostic - 15 hours

Generally, each web page should have a similar look and "feel" like all the others to allow for easy navigation. Roughly speaking, every web page should have a navigation bar, and a similar format

~> Navigation bar

- The navigation bar should have links to the three main pages: the homepage, list page, and account page
    - These links will take the user to their unique homepage, unique list page, and unique account page
    - Guest are disabled to navigate to account management page and shows "Guest" in all the {username} block.
- The bar should be located in the top left corner of the website

~> Further Information on Agnostic Features is contained and intertwined with the features we have mentioned above. We have multiple internal algorithms (for purposes such as sorting list items, assigning items based on priority to cohort members, statistics, and more). These algorithms will be implemented as helper utilities agnostic to each of the website pages.

CSCI 201 Design Document

Hardware + Software Requirements

~> Software:

1. Front End: HTML / CSS, Javascript
2. Back End: Java Version 14, SQL Execution Statements
   SQL Execution Statements used for functions such as inserting an item, inserting a user, inserting a cohort, adding to cohort, removing from cohort, deleting an item, deleting cohort, updating a cohort, updating items, update the users
3. Database: MySQL Database Version 8.0.31 / JDBC driver
   a. Pulled data to send to front end to display relevant statistics
4. External Software Tools: Atlassian Jira Software Project Management, GitHub / Git, Google Suite
   a. Used a 3-tier branch workflow philosophy in Git
   b. Managed each member's tasks through Jira
   c. Wrote documentation on Google Suite
5. WebSockets
6. Communication Tools: Discord, Slack, Zoom
   a. Checking form of communication (discord) at least 3 times a day
   b. Frequent meetings and check ups on progress
7. Internal Software "IDEs": Eclipse, VS Code, MySQL WorkBench

~> Hardware:

1. We will utilize a remote-based cloud server for hosting our web application for the duration of this project. For testing, we can utilize a local server to test basic networking functionality.
2. Computer with Intel Core i9-7900X or better
3. RAM 4 GB

<u>Networking + Multi-Threading Functionality for the Project</u>

We plan to have a single backend server with a set of databases storing all the information. We have support for multiple clients. Clients can be authorized users, authorized users in a cohort, or guest users. Clients can initiate changes and interact with our backend server. We implement each connection and request/response as separate threads. Our backend resources (for the purposes of data mutation) will be a synchronized resource. Implementing multi-threading, this ensures that even with X users / clients, only one can make active changes at a time simply to prevent any data intersectional concerns. Client sessions will connect to the backend server, so that any information displayed on the client screen will be periodically refreshed with any updates from the server. The server must be running at all times but the client sessions can be temporary in nature. All client data will ultimately be sorted in the backend w/ periodic updates so that any client can continue their individualized session. Note that this feature of data and session persistence is not the case with GUEST users. Any guest session (upon termination) will lose any and all information pertaining to the session. Guest data is connected to a separate table in the back-end such that multiple batches of guest information / requests / clients do not interpret the authorized user experience.

Introduction + Networking Explained
Our web application ("DormUtils") hosts a central server with connection support for multiple clients. Our team hosts the server on localhost server. When individuals visit the localhost URL in a browser window, this implicitly constructs a client endpoint.

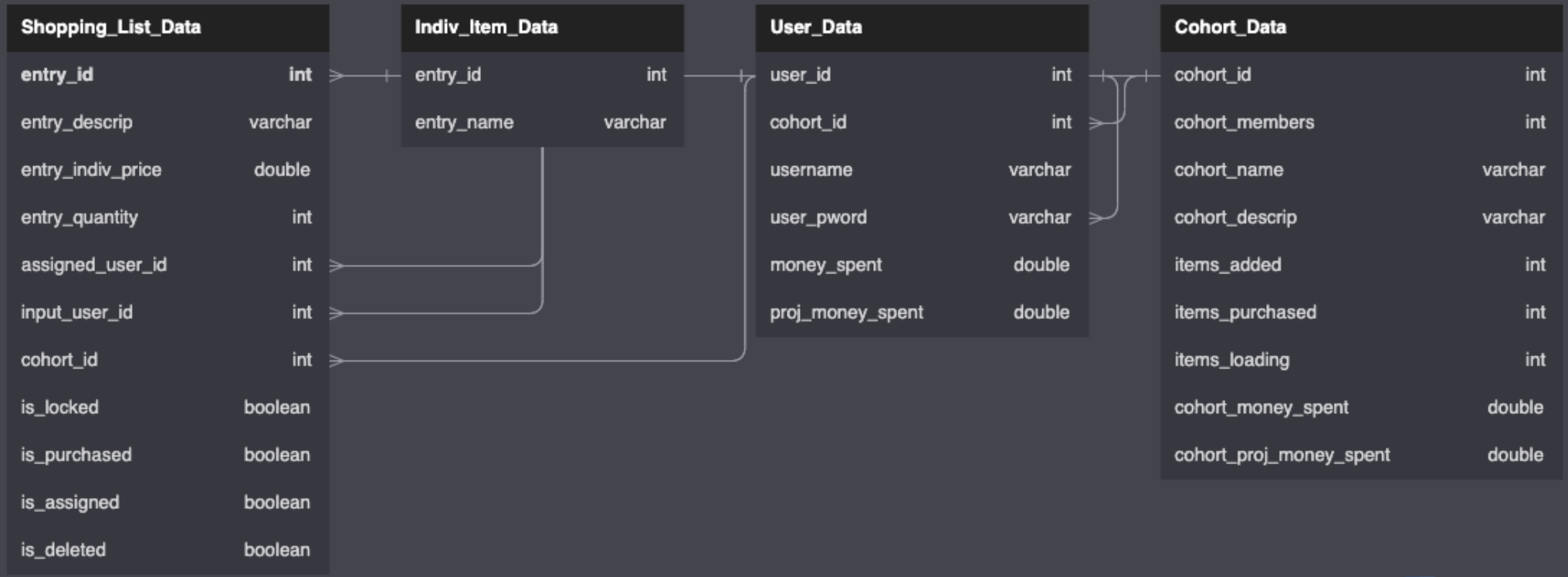
Our team leverages the Websocket protocol to establish a client-server bidirectional communication channel between each client and the server. This channel is persistent and only deconstructs upon termination of a client or server endpoint.

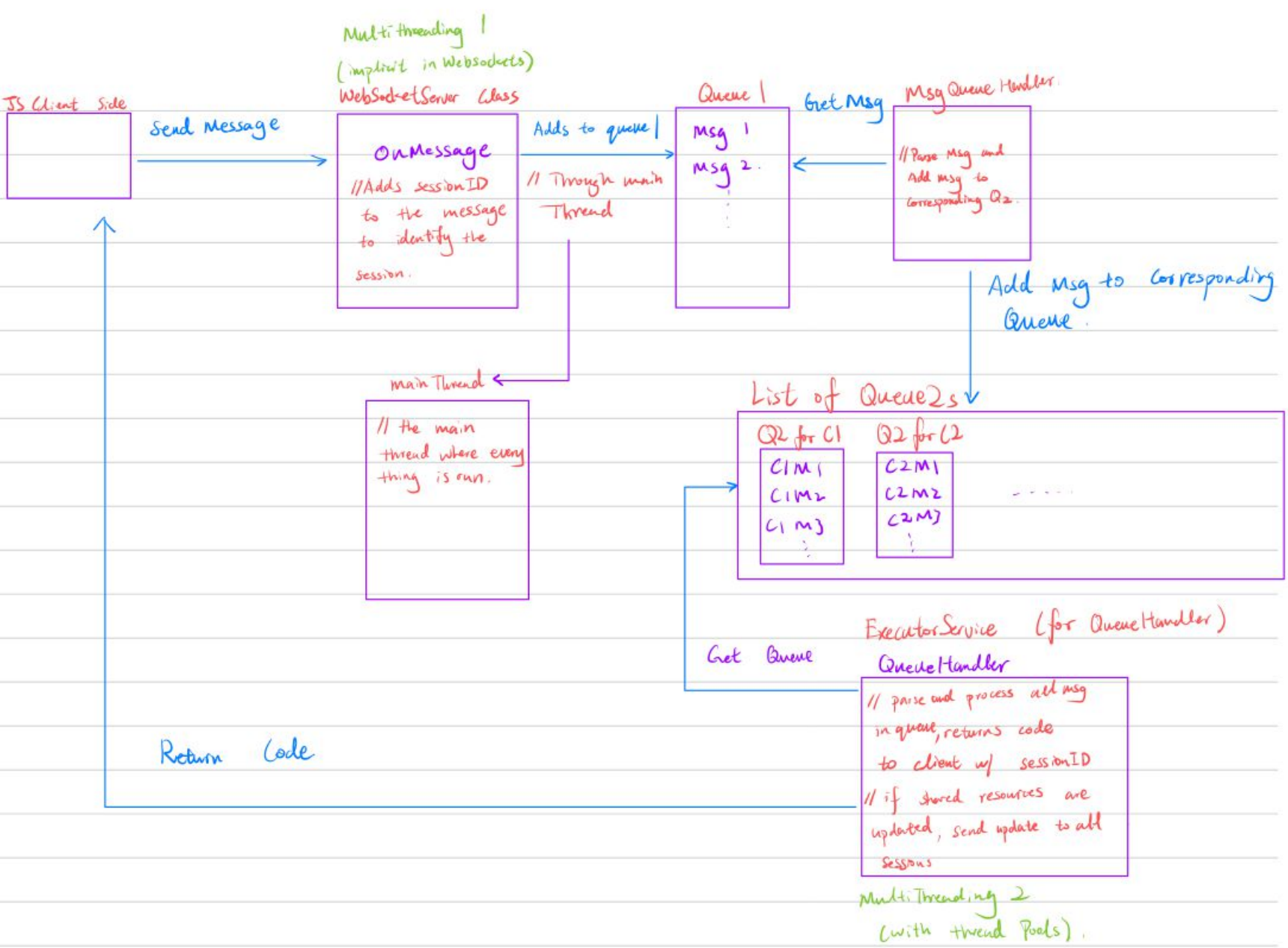
When an "essential" CRUD (create, read, update, delete) action needs to be executed, clients communicate this information to the server in the form of requests. An example of an 'essential' action is (1) registering a new user (2) updating user dashboards or (3) authenticating user login. Nonessential actions can be as simple as static web page navigation **no request to the server is required here**

The following is the generic structure of a client request to the server:
NOTE: {clientID} and {cohortID} of 0 can be interpreted as a {guest / non logged in user} and {not in cohort}
{clientID}~{cohortID}~{functionName}~{functionArgs}
Ex. "0~0~createAccount~username~password"
Ex. "3~7~deleteListItem~itemName"

Once the client request goes through, the server will process the request (see 'Multithreading Section' below) and return a data response to the initial request. Generally, this will update the website in some shape or form (ex. reroute to new webPage) or change data stored client-side.

The following is the generic structure of a server response to a client:
{functionName}~{statusCode} [for get() functions ~> {functionName}~{jsonData}]
Ex. "validateLogin~1"
Ex. "getAllUserInfo~JSON_FOR_USER_OBJECT~JSON_FOR_COHORT_OBJECT"


Shopping_List_Data
entry_id int
entry_descrip varchar
entry_indiv_price double
entry_quantity int
assigned_user_id int
input_user_id int
cohort_id int
is_locked boolean
is_purchased boolean
is_assigned boolean
is_deleted boolean
Indiv_Item_Data
entry_id int
entry_name varchar
User_Data
user_id int
cohort_id int
username varchar
user_pword varchar
money_spent double
proj_money_spent double
Cohort_Data
cohort_id int
cohort_members int
cohort_name varchar
cohort_descrip varchar
items_added int
items_purchased int
items_loading int
cohort_money_spent double
cohort_proj_money_spent double
dbdiagram.io


Multi threading 1
(implicit in Websockets)
WebSocketServer Class
JS Client Side
Send Message
OnMessage
//Adds sessionID to the message to identify the session.
Adds to queue
// Through main Thread
Queue 1
Msg 1
Msg 2.
Get Msg
Msg Queue Handler
// Parse Msg and Add msg to corresponding Q2.
Add Msg to corresponding Queue.
main Thread
// the main thread where everything is run.
List of Queue2s
Q2 for C1
C1M1
C1M2
C1M3
Q2 for C2
C2M1
C2M2
C2M3
ExecutorService (for QueueHandler)
Get Queue
QueueHandler
// parse and process all msg in queue, returns code to client w/ sessionID
// if shared resources are updated, send update to all sessions
Return Code
Multi Threading 2
(with thread Pools).

Backend Structures Class Diagram

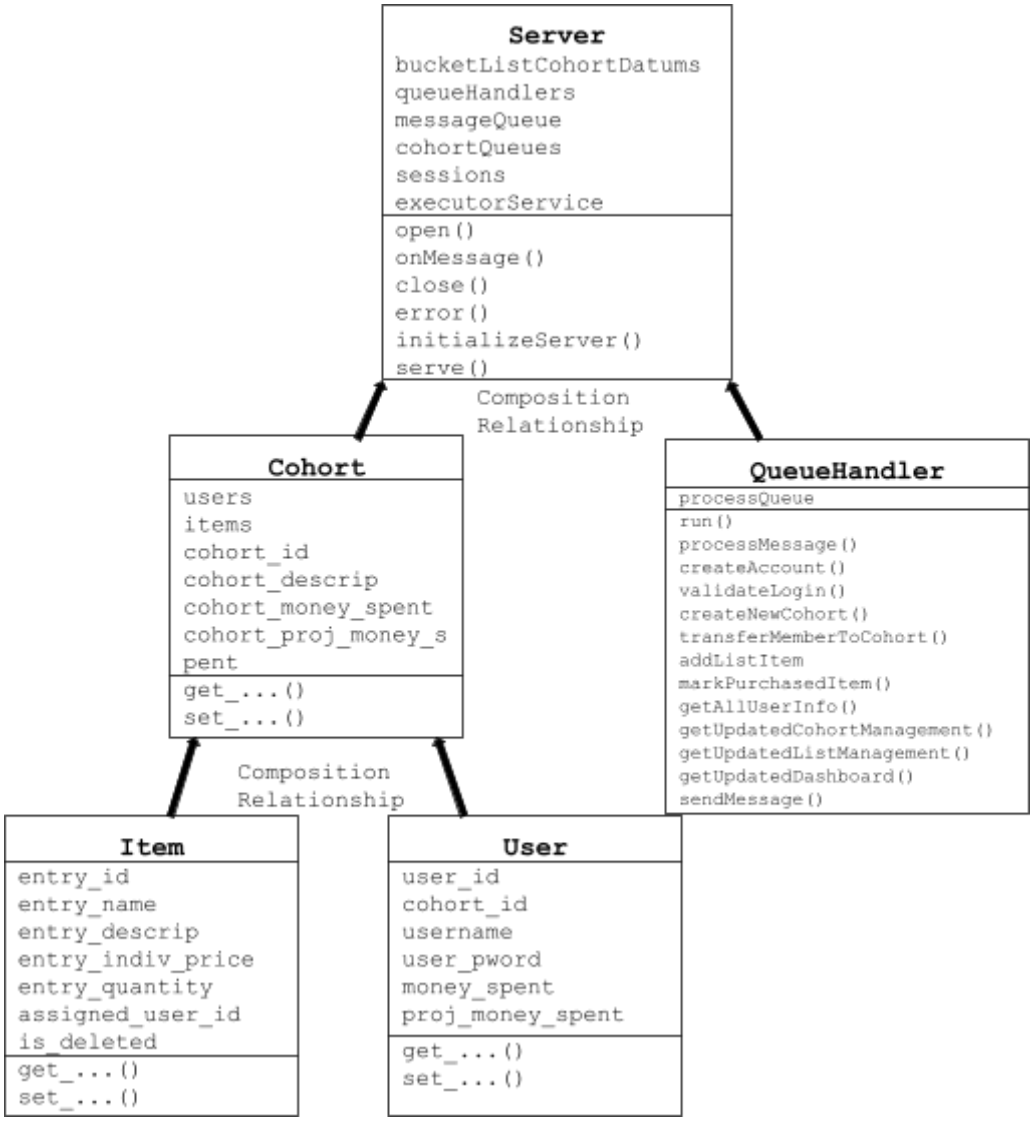


Diagram Explanation:

The Cohort class contains a list of Users and a list of Items. The class also contains several other member variables that describe statistics concerning the cohort.

The Item class contains all of the information pertaining to an individual item..

The User class contains information pertaining to each user. The data member money_spent is how much the user has actually spent, while the data member proj_money_spent is how much the user is projected to spend once the user buys all the items assigned to them.

The QueueHandler class contains all functions that modify the Cohort, Item, and User classes. This class reads a statement or request from the frontend and updates the backend accordingly. QueueHandler extends from thread, so multiple QueueHandlers can run concurrently.

The Server class holds a list of Cohorts. These Cohorts are effectively an object copy of the SQL database. Rather than interact with the SQL database directly, the frontend instead interacts with this copy, minimizing retrieval time. The SQL database will be updated periodically, rather than after every operation. The server class also contains the QueueHandler threads that will process each message using multi-threading.

SQLUtil Class Diagram

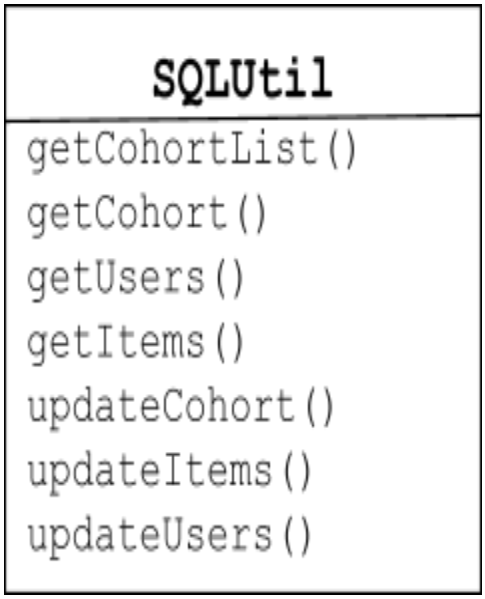
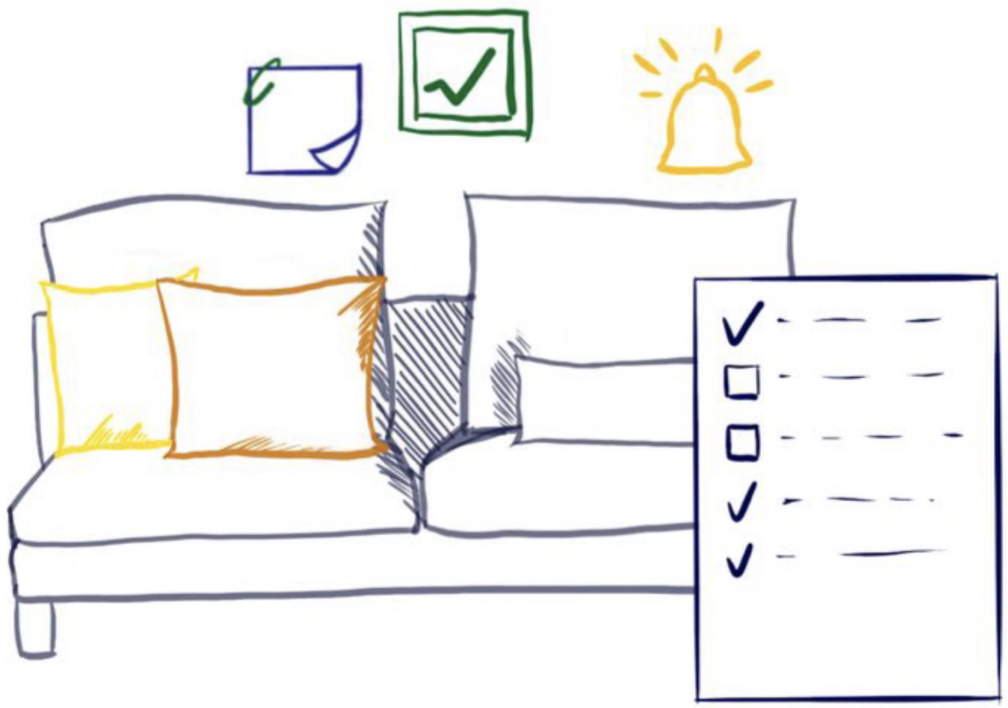


Diagram Explanation

The SQLUtil class interfaces with the database, allowing the backend to read from and write to the SQL tables. When a client interacts with the website, the data is sent to an object copy of the SQL database, which is created using getCohortList(). Periodically, updateCohort() will be called on every Cohort Object in the backend, ensuring that the SQL database remains synced with the object copy.

[NAME] - [DESCRIPTION]



Sign up | Log in

Welcome Back!

username

password

Validation error displays here...

LOG IN

Continue as a Guest

HELLO,

Username

email@address.com

Dashboard

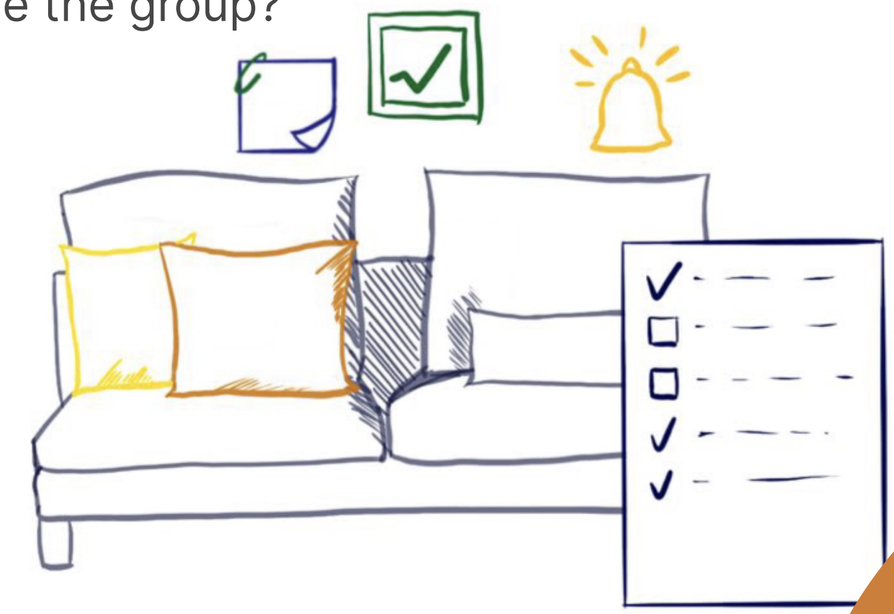
Displayed List

Log Out

You’re in the group, [GROUP_NAME]

- ☐ Group ID: [Group_id]
- ☐ They are in the same group with you:

- ☐ Do you want to leave the group?
  - ▶ Yes.
  - ▶ No.



HELLO,
Username
email@address.com

Dashboard

Displayed List

Log Out

Oops, you're not in any group now...

☐ Do you want to join an existing group?

Enter the group ID:

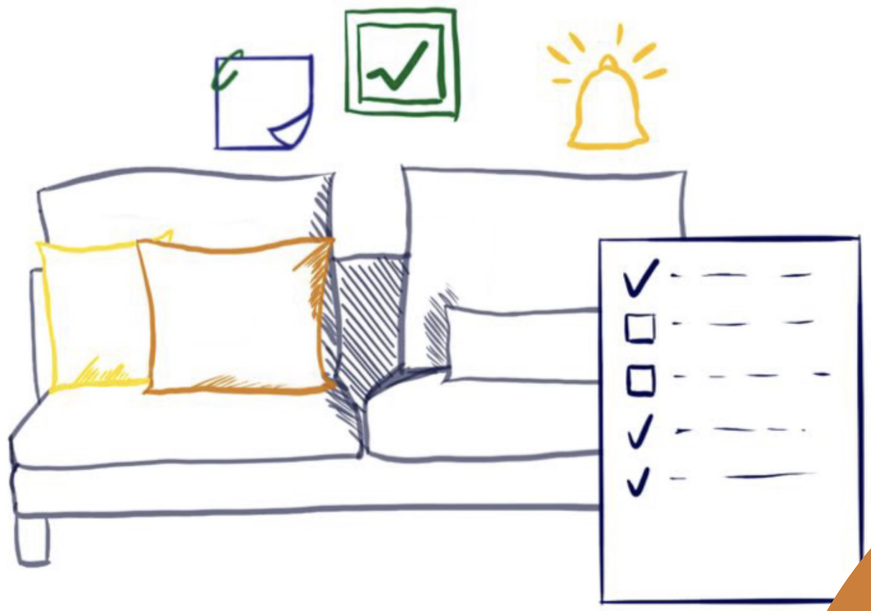
Join

Group ID error display...

☐ Do you want to create a new group?

▶ Yes.

▶ No.

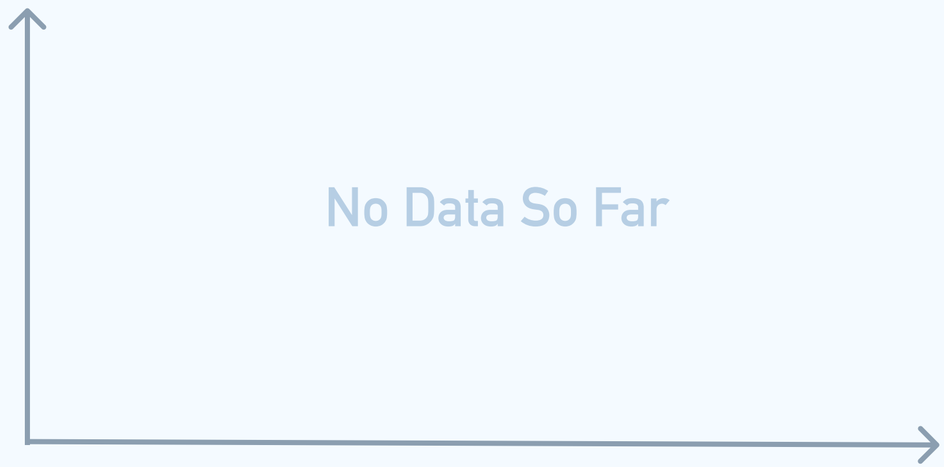


Don’t have a group?
Join / Create one here

Shopping Cart

Item	Amount	Where	Deadline	Done?
Congratulation! You’ve completed all your tasks!				// Cross-out completed purchase tasks here

Statistics



Total Cost:
$0.00

Shopping List

Item	Amount	Where	Deadline
The List is Empty. Add an item here...			

[GROUP_NAME] Shopping List

with [MEMBER_A],[MEMBER_B],...

ADD

ITEM: _____ DEADLINE: _____

QUNATITY: _____ BUYER: _____

PRICE: _____ WHERE: _____

ETRA DESCRIPTION (optional):

ITEM	QUNATITY	PRICE	DEADLINE	BUYER	WHERE	NOTES

// The table content could be scroll down

Don’t have a group?
Join / Create one here

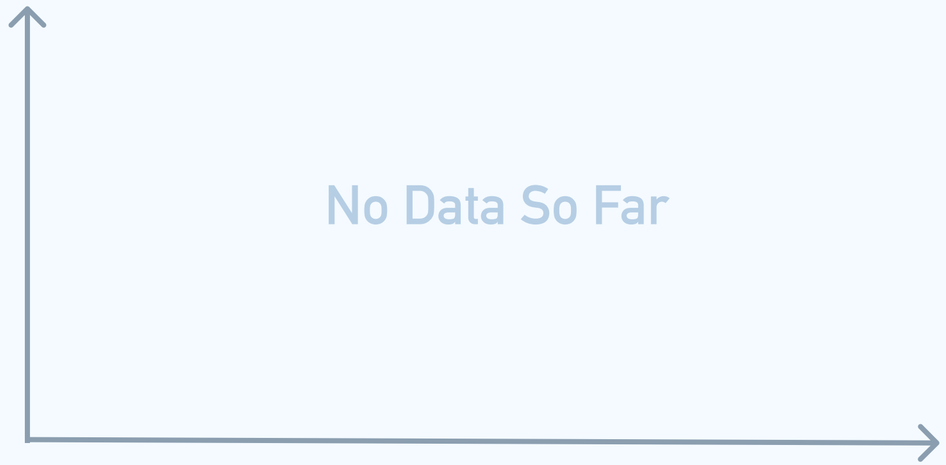
Shopping Cart

Item	Amount	Where	Deadline	Done?

Congratulation!
You’ve completed all your tasks!

// Cross-out completed purchase tasks here

Statistics



No Data So Far

Total Cost:
$0.00

Shopping List

Item	Amount	Where	Deadline

The List is Empty.
Add an item here...




[GROUP_NAME] Shopping List

with [MEMBER_A],[MEMBER_B],...

ADD

ITEM: _____ DEADLINE: _____

QUNATITY: _____ BUYER: _____

PRICE: _____ WHERE: _____

ETRA DESCRIPTION (optional):

ITEM	QUNATITY	PRICE	DEADLINE	BUYER	WHERE	NOTES

// The table content could be scroll down

We are given a list of users.
We are also given a list of items.

Design an algorithm capable of assigning (unpurchased, unlocked, unassigned) items (0 to X) to users in a manner that minimizes the variance between proj_money_spent for each user. (aka minimizes the max proj_money_spent of any user).

Items have a (1) price (2) quantity (3) “locked” property (4) “purchased” property and (5) “assigned” property.
~> The list of items can have items added to it and removed from it between executions of the algorithm. Each change to the list of items necessitates a new execution of the algorithm to be run. Every time the algorithm is run, it will assign certain items to certain users in a manner that ensures that the proj_money_spent of each user will **minimize variance** across the board.
~> Users have the ability to mark off an assigned item as “purchased” once the item has been purchased. When this action is completed, it prevents the item from being reassigned in any future iterations of the algorithm and is permanently added to the money_spent amount for the user that purchased the aforementioned item.
~> Locked is a boolean value. After an execution of the algorithm, users have the option to “lock” an item that is assigned to them. This means a future execution of the algorithm will NOT change the assignment of this item to this user. Any “unlocked” items are subject to change upon future executions of the algorithm with the addition or removal of new items to the list of items.

Users have a data member (1) money_spent and (2) proj_money_spent
~> money_spent is the money a user has spent on PURCHASED ITEMS to date
~> proj_money_spent is the money a user is projected to have spent that consists of (1) money_spent on PURCHASED ITEMS + (2) the cost of any unpurchased, “locked” items the user has assigned to themself and (3) the cost of any “unpurchased, unlocked, assigned” items for a user as well

// how I view this problem: the algorithm will consist of two parts, where one takes care of “locked” property and all those stuff, and the other does the actual assignment, assuming the previous part has already taken care of all these “edge cases.” Then, the second part reduces to my little array problem.

Unit Testing - Registration / Password

Test Case Type	Description/Test Goal	Test Step	Expected Result
Registration	Ensure that registration communicates with the database.	Register with a valid username and password.	User becomes logged in, and the username and password exist in the database.
Registration	Ensure registration uniqueness.	Input a non-unique username while registering.	No valid registration will occur and the database will remain unchanged.
Login Security	Verify password-username rules are working.	Input a valid username and password.	Successful login, user redirected to dashboard page.
Login Security	Validate password input.	Input a password with a character that is non-alphanumeric and not a special character.	No valid login will occur, and the user remains on the same page.

Unit Testing - Cohort Management

Test Case Type	Description/Test Goal	Test Step	Expected Result
Cohort Creation	Ensure that cohort registration communicates with the database.	Create a cohort while not currently in a cohort.	A valid cohort will be created with a unique cohortID.
Cohort Creation	Test parameters for cohort creation.	Create a cohort while currently in a cohort	A valid cohort will not be created, and the database will remain unchanged.
Cohort Security	Verify cohort password rules are working.	Input a valid cohort password while currently not in a cohort.	The user will be added to the cohort.
Cohort Security	Verify cohort password rules are working.	Input a invalid cohort password.	The user will not be added to the cohort, and the database will remain unchanged.

Unit Test - List Integrity

Test Case Type	Description/Test Goal	Test Step	Expected Result

List Modification	Test integrity of add function.	Add an item to the list with valid parameters	A new item appears on the list page, and the database has an additional item for this cohort.
List Modification	Test integrity of remove function.	Remove an item from the list.	List no longer displays this item on the user's list page or any future instances of the list page.
List Modification	Test list input validation	Add an item to the list with missing vital information (missing name/quantity/price)	No new item is added to the list, and the database remains unchanged.

Unit Test - UI

Test Case Type	Description/Test Goal	Test Step	Expected Result
Navigation	Verify link functionality	Press a link in the navigation menu (dashboard / list page / account management).	The link brings the user to the corresponding webpage (dashboard / list page / account management).
Display	Verify display size and scaling	Navigate the website with a variety of screen sizes and aspect ratios.	The website remains functional regardless of screen size.

Unit Test - Internals

Test Case Type	Description/Test Goal	Test Step	Expected Result
Internal Algorithm Validity	Verify functionality of the algorithm.	Add an item to the list.	The algorithm correctly assigns a user to an item.
Internal Algorithm Runtime	Verify the runtime of the internal algorithm	Assign a large number of items to a large number of users simultaneously.	The database will be updated near-instantly, and the server will not lag.

Unit Testing - Guest Management

Test Case Type	Description/Test Goal	Test Step	Expected Result

Guest Session Logistics	Check that guest sessions are created.	Login as a guest.	The database will reflect a new guest session and handle its data differently than with authorized users.
Guest Session Logistics	Check that guest data is inaccessible after the session closes.	Leave the website after inputting in data as a guest. Navigate back to the website as a guest and view a user-specific web page.	The web page will reflect a new guest session, and the old guest data will be inaccessible.
Guest Session Logistics	Ensure that guests cannot interact with cohorts.	Try to join/leave/create a cohort as a guest.	The guest will be unable to perform any cohort-involved activities, and the database should remain unchanged.

Component Testing

Test Case Type	Description/Test Goal	Test Step	Expected Result
Login-Dashboard Communication	Verify that the login communicates with the dashboard.	Login and view the dashboard.	The dashboard page will be updated to reflect the user's unique data.
Cohort-Dashboard Communication	Verify the cohort page communicates with the dashboard.	Join a cohort and view the dashboard page.	The dashboard list will display the user's cohort members and list items.
Session Memory	Ensure that a user stays logged in when refreshing the page.	Refresh any web page while logged in.	The user will remain logged in.

System Testing

Test Case Type	Description/Test Goal	Test Step	Expected Result
Multithreading / Networking	Check that users cannot access synchronized materials at the same time.	Perform operations on the shared lists at the same time. (Add to the item list or modify the cohort member list).	The lists will maintain their structure and no data will be overwritten. The operations will execute as if they had been performed sequentially.

Multithreading / Networking	Ensure that requests are processed in a timely manner.	Send multiple requests from different clients at similar times.	The requests will be processed in the order they were submitted, ties broken by userID.

Performance / Stress Testing

Test Case Type	Description/Test Goal	Test Step	Expected Result
Cohort Response Time	Test the speed of cohort creation and joining.	Try joining a cohort that was just created.	The user can join the cohort after a very slight delay (seconds) from the cohort's creation.
Registration / Login Response Time	Test the speed of registration / login.	Have a multitude users register / login simultaneously	The registration and login times should be relatively unaffected.

CSCI201 Group 4 Deployment Plan

OVERVIEW

There will be two main methods of accessing our final project.
Method One ~> User Access
Method Two ~> Developer Access

BACKGROUND INFORMATION

As a team, we have performed certain tasks and steps throughout our development process in this project to aid in deployment.

(1) Git Workflow ~> We maintain a github organization with multiple repositories to document changes to our code. We implemented a three (3) tiered branch system as we develop to ensure that multiple people can work and update their code at the same time without conflict and we can maintain a production git log to ensure proper integration before publishing our main branch. We have three branches:
   (a) a main (master) branch ~> the final iteration of the code will be found here. This is the most updated * safely integrated * code used in the live version of the web application we will host.
   (b) a developmental (testing) branch ~> this is a branch which is continuously updated by the addition of completed features. Integration is performed as developers submit pull requests and merge feature branches here. This is a testing branch so we can test recently added code and make sure our website does not break under new features.
   (c) multiple feature branches ~> every time a subteam of developers start working on a new feature, they create a new branch to work on the feature in isolation without updates from other substeams. Upon the completion of this feature, they merge the current feature branch with the develop branch and perform subsequent integration tests.
   (d) Importance: The importance of maintaining this git workflow is that if we ever encounter any issues in the deployment or lifetime access of this web application, we can always have a paper trail of organized git commits and branches to backtrack through to determine where the issue truly is located. Maintaining this git workflow promotes future maintenance and a smoother deployment process upon each update to the web application.

User Access is intended for use by the general public or any individual who desires to use the application for its functionality. In order to deploy our project for User Access, the following steps will need to be completed:

(1) Ensure that integration is complete (to the best of our ability + after running tests)
(2) Make sure that all external dependencies are correctly configured, added, and linked.

(3) Push the application to a server on Google Cloud as suggested by our CSCI 201 professor. If Google Cloud is not available, then run it with a Microsoft Azure server available for integration through github. After push, check to ensure that integrity of the code, markup files, documentation, and dependencies.
(4) Ensure that all necessary data was migrated to the live server from the existing system and run unit tests to verify the integrity of the SQL database.
(5) Perform a full regression test of the website. Include both clients from the same laptop as well as clients from multiple laptops and initiated from different WiFi systems.
(6) Re-Run previous tests (functional and non functional) to ensure that previously developed features still perform as expected. Perform alpha tests to check on the user experience with the uploaded database.
(7) Notify the users of the updated application and provide them with the public URL. Utilized advertising and marketing services (such as ad personalization or mass emailing) to target users on the web who would be interested in our product. Perform a beta test on the notified users.
(8) Ensure that current users are satisfied with the application and provide periodic ways of receiving user feedback.

Developer Access is intended for use by a select group of developers who want access to the files and structure behind the application itself (i.e. the code and any build files). In order to deploy our project for Developer Access, the following steps will need to be completed:

(1) On our git repository, we also have patch-based releases. Every time a new major feature is added to the code, we bundle up all the code and dependencies (in the future, will be managed by maven) into a release package for easy download. Additionally, read the above section on *git workflow.*
(2) Largely, our entire development process is stored on Atlassian Jira Software as project reports or in our backlog. Identified there is which developers worked on which tasks as well as time estimates and task breakdowns. This is useful for future deployment of similar projects or when estimating time for updated deployments of the current web application.

CURRENT DEPLOYMENT PROCEDURE

Open the server on localhost tomcat.

- Open the project on Eclipse IDE, and open the server on localhost.

Test if there is a connection issue.

- Run a basic javascript file to test for connection between front end and back end web sockets.

Enter the URL of the websocket for the client code

- Modify the front end files to connect to the server IP, rather than localhost. This requires both the client and server to be on the same internet connection.

Send the newest client code to the user.

- Test the server functionality with the full, proper client code

Finalize testing and deploy

PACKAGES REQUIRED FOR DEPLOYMENT

JDBC, GDON

STEPS FOR RUNNING BACK END PROJECT FILE

1. Import jar file into Eclipse, JDBC and GSON packages will be included in the jar file.
2. Open MySQL and run the SQL server on the back end server.
3. Update the SQL url and password in SQL utils.
4. Run the Database code in MySql.
5. Run the Server class to begin.