RoomEase

Design Document

02.08.2025

Team 16:

- Krish Patel
- William Parodi
- Nathan Huang
- John Raub
- Ishan Muhkerjee
- Carter Ellis

Index 2

Purpose:

The purpose of RoomEase is to revolutionize the way roommates coordinate and collaborate in shared living spaces by providing a centralized, all-in-one solution for their daily challenges. Designed with the understanding that managing schedules, quiet hours, chores, and shared expenses can often lead to unnecessary stress and miscommunication, RoomEase empowers users to streamline these tasks seamlessly. By creating roommate groups, roommates can efficiently share chore responsibilities, maintain grocery lists, organize shared calendars, and even track expenses—all within a single platform. Unlike existing apps that focus on individual aspects of roommate life, RoomEase consolidates these essential features into one intuitive application, fostering better communication, reducing conflicts, and enhancing harmony in shared households. With RoomEase, roommates can focus less on logistics and more on building an enjoyable living environment.

Functional Requirements:

1. User account registration:

   As a user,

   a. I would like to register and login using my Google account.
   b. I would like to log out of my account and save my data.
   c. I would like to be able to create a profile with my name, birthday, profile picture, and contact information.
   d. I would like to be able to edit my profile at any time.

2. Profile management and user settings:

   As a user,

   a. I want to be able to customize the appliances within each individual bedroom.
   b. I want to be able to set chore difficulty levels and allocate points based on difficulty.
   c. I want to add notes/comments/expectations on chores (when completed / how to complete).
   d. I want to be able to customize the “master bedroom” so that each different group has their own customizable aesthetic.

3. Recurring expenses:

   As a user,

   a. I want a UI to display all bills/ recurring expenses in the group.
   b. I want to add bills/recurring expenses onto the bills UI and be able to indicate which group members are a part of the expense.
   c. I want to edit any of my bills/recurring expenses from the group and be able to edit which group members are a part of the expense.
   d. I want to indicate a “Pay Master” (the person that actually pays the bill) for each bill/recurring expense that everyone must send what they owe to.

e. There will be a “Pay Master”, I want to indicate when members of my specific bill/recurring expense have paid me or not.
f. I would like to see a history of how much recurring expenses have cost in the past (track changes in electricity/water usage).
g. I want to receive web notifications when a new bill or recurring expense has been created with me as a contributor.
h. I want a UI to display my total balance owed/am owed by each group member.
i. I want to mark as settled all of what I owe/am owed by each group member at once (this would mark all payments requested as “paid”).

4. Quiet Hours and Room State:

   As a user,

   a. I want a UI that displays all room states requested (present and future).
   b. I want to view the room’s current state.
   c. I want to set quiet hours for the household.
   d. I want to receive web notifications when a roommate’s requested quiet hours has begun or been scheduled.
   e. I would like to be able to see a clock which dictates the current state of the room

5. Creation and selection of groups or rooms:

   As a user,

   a. I want to launch a main page that displays all my roommate groups.
   b. I want to create and join a roommate group.
   c. I want to be able to invite others to my roommate group.
   d. I want each roommate group to have a UI that displays all the features that RoomEase provides for each group in the form of common appliances.
   e. I want a room called the “master room” which will compile all the tasks, quiet hours, expenses from all groups that I am a part of into one room UI.

f. I want a settings page for each roommate group where I can enable or disable certain RoomEase features such as grocery splitting, chore list, etc.
g. I want to be able to leave my group and notify other members in the group.

6. Handing of chores and tasks:

   As a user,

   a. I want to assign, track, and set deadlines for household chores.
   b. I want to receive web notifications when a chore is due or completed.
   c. I want to mark my chores as complete.
   d. I want a feature that allows me to request to swap chores with one of my other roommates

7. Maintaining group grocery lists:

   As a user,

   a. I want to add items to a shared grocery list.
   b. I want to add comments to the items I put on the list.
   c. I want to mark an item on the list as purchased and request payment for the amount that I spent on it.
   d. I want to be able to mark when a payment has been completed.
   e. I want to receive web notifications when I have unpaid grocery balances.

8. Creating a communication network:

   As a user,

   a. I would like to have a group chat for each roommate group.
   b. I would like to enable a chat filter (blur profanities) for my roommate group chat.
   c. I want to upload a group photo that will display on the door that leads to the group page.
   d. I want to enable a bulletin board in my room where I post and remove notes to be publicly displayed.
   e. I would prefer to create and see clauses in my roommate agreement with my roommate(s).

f. I want to be able to see and add house rules to a section of the bulletin board.
g. I would like to add pictures to a section of the bulletin board to save memories that I share with my roommate.
h. I would like to be able to interact with the different sections of the room that show me the different features.

9. Settling disputes among group members:

   As a user,

   a. I want to be able to click a gavel on the desk that allows roommates to vote on disputes within the apartment
   b. I want to be able to bring up a page to vote on disputes
   c. I want to be able to issue a dispute through the UI the gavel brings up when clicked.

10. Implementation of points system:

    As a user,

    a. I want to be able to earn points based on completing my tasks/getting groceries for my roommates.
    b. I want to use points to purchase cosmetics to decorate my rooms.
    c. I want to be able to enable a monthly review of my roommates in certain categories (cleanliness, noise, expenses paid, etc.) for groups larger than 2 people.

11. Leaderboard management:

    As a user,

    a. I want to see “RoomEase Wrapped”, which is a monthly leaderboard of who did the most of what chores / bought groceries the most, etc.
    b. I want to view an all time, dynamically updated leaderboard of who did the most of what chores / bought groceries the most, etc. throughout the entire existence of the roommate group.

Non-Functional Requirements

1. **Architecture and Performance**

   We plan to develop an app with a separate frontend and backend that is connected with RestAPIs. This will allow the team to separate work better and update each side separately.

   The backend will be developed using Express.js, which will handle API requests and communicate with a MongoDB database to store and process data. The backend will expose RESTful API endpoints to deliver relevant data to the frontend. We expect our API response times to be under 200ms for simple queries and under 500ms for more complex database operations under normal load conditions. The frontend will be developed using React.js that will make requests to the API to populate the application's visuals with relevant information. Using React will also allow us to port the application to multiple different devices (Android, IOS, Web)

2. **Security**

   We plan to utilize the Google API features so that users must login through their google accounts. This will allow us to use the built-in google security features to prevent users from accessing other accounts. Additionally, we plan on screening user inputs to ensure that they are valid. This will allow us to avoid XSS and code injection attacks as users won't be able to upload malicious items.

3. **Usability**

   Usability is a top priority for RoomEase. We will create a simple and easy to navigate UI that clearly displays the most important information to the user at any point in time. We additionally aim to limit the amount of pathways that a user must go through to access any specific information they are looking for. Many

other applications over-complicate their system by creating many channels that often go almost unused. We will mitigate this by organizing functionalities into groups so that a user can clearly navigate from one task to another in as few clicks as possible. We aim to have 5 or fewer clicks from the homepage to any individual feature that the user is attempting to access.

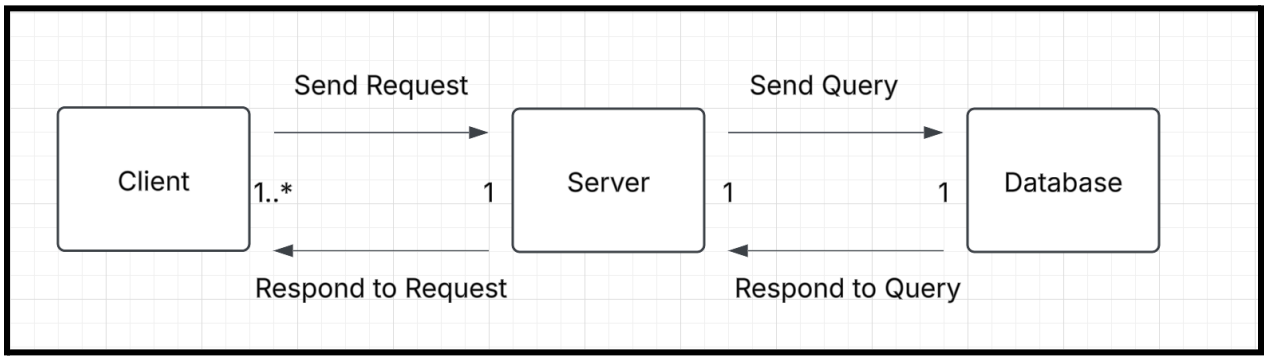
4. **Hosting/Deployment**

   Since the backend and frontend are disjoined, they can be deployed and updated separately. Further down the development process, we can host the backend on a free Render server for testing. The frontend can then be hosted as a Github Pages website.

Design Outline

High Level Overview

RoomEase is a web application designed to help roommates coordinate chores, expenses, and schedules efficiently. The application follows a client-server architecture, where the frontend communicates with the backend through API requests. The backend processes these requests, interacts with a MongoDB database to store and retrieve user data, and sends appropriate responses to the frontend. This structure ensures a seamless user experience for managing shared household responsibilities.



1. Client

a. The client provides users with an intuitive interface to interact with RoomEase.

b. The client sends HTTP requests to the server through to fetch or update data related to chores, expenses, and schedules.

c. The client processes server responses and dynamically updates the user interface to reflect changes.

2. Server

a. The server receives and handles API requests from the client.

b. The server validates requests and communicates with the database to store, modify, or retrieve

roommate-related data.

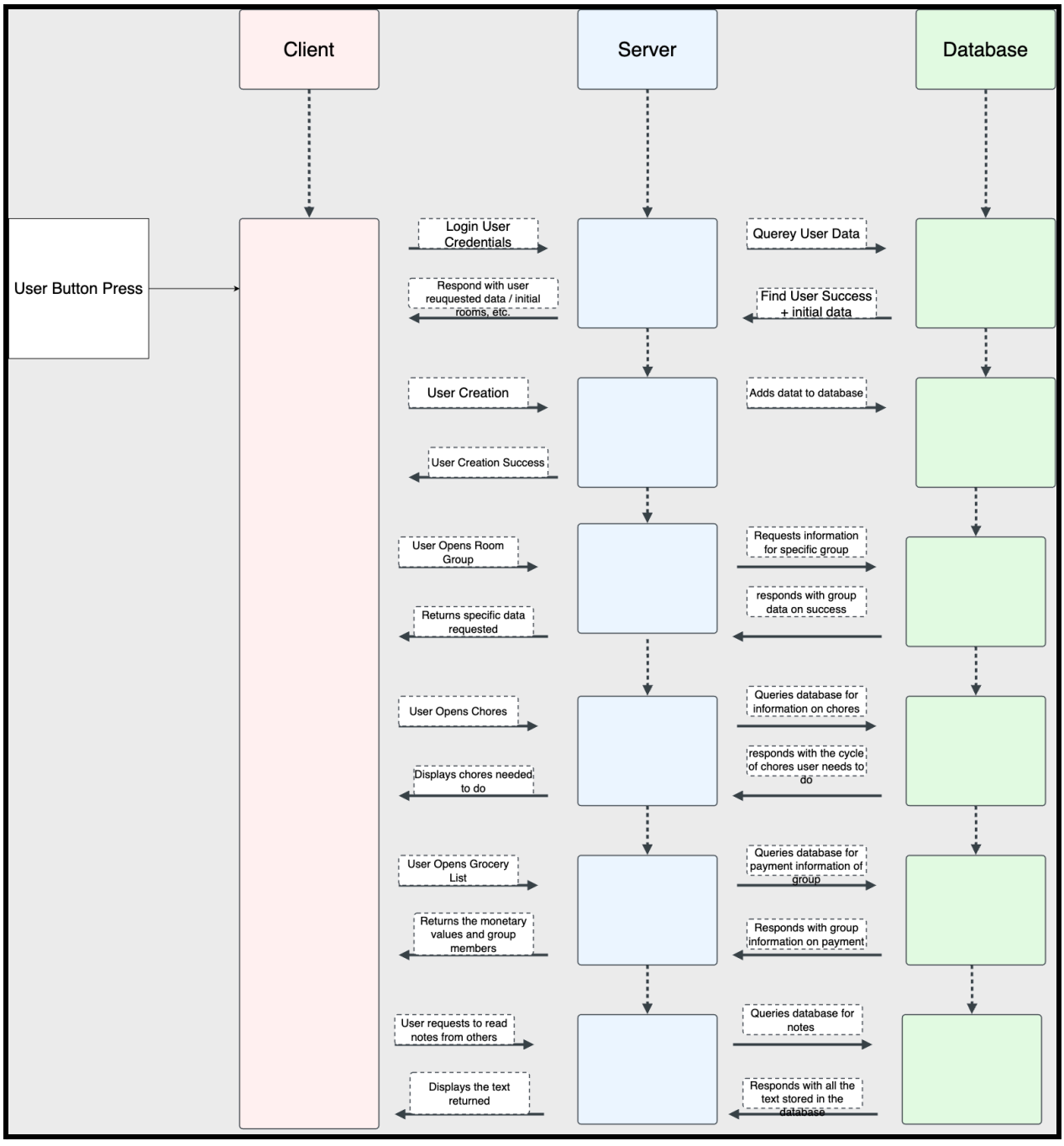
c. The server generates appropriate responses and sends them back to the client.

3. Database

a. A MongoDB database stores all relevant application data, including user profiles, chore assignments, expenses, and schedules.

b. The database processes queries from the server and returns the requested data efficiently.

Sequence of Events Overview



The sequence diagram for RoomEase illustrates the typical interaction between the client, server, and database. The process begins when a user launches the application and attempts to log in. The client sends a login request to the server, which processes the request and queries the database for user authentication. Once authenticated, the client can send additional requests to

the server to perform actions such as managing chores, updating expenses, and coordinating schedules. For each request, the server interacts with the database to retrieve or update information. The database processes these queries and returns the requested data to the server, which then responds to the client, ensuring real-time updates and a seamless user experience.

Design Issues

Functional Issues

1. How are we going to handle logins, how will users create an account and register?
    - Option 1: Don't require users to login
    - Option 2: Allow users to create their accounts using an email and password which is then stored on the RoomEase database
    - Option 3: Have users create their accounts through their Google accounts and then use the Google API for logins.

Chosen Solution: Option 3

Discussion: Not having users log in wouldn't allow us to implement many of the important features that roommates want as creating groups would naturally require some login system. For those reasons, the group opted against option 1. In discussions of whether or not to implement our own login system or to allow users to log in through Google, we talked heavily about the security features and what would make our users feel the most comfortable. On one hand, we felt that creating our own login system would allow us to have everything handled on our side, and not create any dependency on other servers to function properly. Additionally, then we wouldn't have any issues with Google creating an update or changing their terms which could alter the functionality of our program. However, we also felt that most users would place greater trust in a Google-based login system since everyone knows Google and it would be far easier and better for us to build on the existing security that Google provides. We also thought that using Google's API would allow us to expand the project later on and potentially incorporate the Google Calendar app for quiet hours scheduling, and use the Google system for notifications. Ultimately, we opted to utilize the Google API for logins since we would benefit from their built-in security. We also noted that the risk of a major change impacting our program was limited because many apps rely on Google's login services so there would be drastic ramifications if Google were to make a change so it is safe to say they won't do much to destroy so many platforms.

2. What form of a communication system should be built into a room?

- Option 1: No chat should be implemented
- Option 2: A group chat should be available, but no direct messaging option
- Option 3: Both a group chat and direct messaging system should be available.

Chosen Solution: Option 2

Discussion: This topic was a huge area of debate for a while because there was a lot of uncertainty as to how much value this feature would add for a typical user. On one hand, we considered not doing a chat at all since most roommate groups would already have a means of group communication setup. However, we also acknowledged that many times roommates could fill their group chats with information that isn't necessarily important, but more of just friendly banter. Many of us agreed that our own personal roommate chats get clogged with "memes" and content of the like. For that reason, we decided to have a group chat feature so that rooms could have relevant conversations saved in the app, and they could have other conversations externally if they chose to do that. For that reason, we opted against option 1. However, we then debated the option for users to send direct messages to each other. In this case, we opted against having a built-in communication system because a roommate group is likely to have everyone's contact information, and the reason for the built-in group chat doesn't apply as well to direct messages because there aren't as many situations where an important conversation between two people would be interrupted. After all, there aren't any other members involved, so the odds of the conversation shifting off-topic seem less likely. Additionally, we felt that storing chat messages for every possible pair of members in a room would drastically increase the space that a room would use up. For that reason, we decided to only have a single group chat feature and no direct messaging capabilities.

3. How can we display/add quiet hours/room state times to the roommate group?
    - Option 1: Calendar View
    - Option 2: A simple queue of current and upcoming events

Chosen Solution: Option 2

Discussion: This topic sparked some debate amongst the group that was resolved fairly quickly. Since our application's main goal is to consolidate the features of other apps roommates may use, including calendar apps, it seemed logical to implement a Calendar View at first, but it became apparent that this UI would make the app feel very cluttered. Implementing a Calendar View would be inefficient as calendars are meant to store large quantities of many different types of

events across many months, this is not necessary for roommates simply requesting quiet hours that are often infrequent outside of sleeping time. Thus, we decided that the second option would provide a simple way to display which events/quiet hours are approaching without having to navigate a whole calendar.

4. How will the features (grocery splitting, bill splitting, quiet hours, etc.) be organized on the screen for each roommate group?
    - Option 1: A vertical, scrollable list that the group members can select in order to initiate the UIs of the features.
    - Option 2: A grid that displays each feature with a logo that relates to the feature (grocery bag logo for grocery splitting, judicial gavel for the dispute polls, etc.)
    - Option 3: A UI that shows the display of a room that contains appliances/objects that correspond to each feature (fridge for grocery splitting, calendar for setting and viewing quiet hours, etc.).

Chosen Solution: Option 3

Discussion: When looking at these options, the two simpler options (vertical list and grid) seemed to be the most enticing as we would be able to implement it quickly and focus on other more complicated portions of the app, but we realized that, by utilizing the room UI style, we would be able to introduce more elements like cosmetics and vibrant displays to improve retention to the app. This strategy would allow us to organize the application in a way that is more intuitive and engaging for the user. Thus, we have decided that Option 3 would be the best option moving on.

5. What would we like our homepage to look like?
    - Option 1: The homepage should contain a list of the different groups that the user is a part of
    - Option 2: The homepage will contain summary information for each room so that a user can see all their responsibilities from the home page
    - Option 3: The homepage will contain doors to each room, and there will be one door for a master room that displays your responsibilities to each room.

Chosen Solution: Option 3

Discussion: Option 2 was our initial idea, but then we decided that it could be overwhelming for a user to immediately see everything that they are expected to do immediately when they logged

in. Additionally, we felt that this implementation would place a lot of strain on the server since immediately upon loading in a user would be requesting knowledge about all of their responsibilities. However, when discussing option 1 we realized that users might actually want to have information on all of their chores at the same time, and having the ability to access a feature for all rooms at once would be beneficial for tracking finances since a user could then see all the money they owe. For that reason, we decided to merge the first two options by still having the display show each room, but then also a path to the "master room" where users could see every responsibility that they have across all of their rooms. Therefore, we get the benefits of not having to load everything immediately, while also giving users the access to everything in one place.

6. How should RoomEase handle roommates paying each other for shared expenses
    - Option 1: We implement our own payment system designed to handle secure payment transfers
    - Option 2: We utilize Venmo's API features to create a system for users to pay their roommates through Venmo
    - Option 3: We create a "mark as paid" button and add an additional system to consolidate all money owed across expenses so users can choose to pay each other how they want to, and handle multiple expenses at once

Chosen Solution: Option 3

Discussion: We almost immediately decided against option 1 because we were very concerned about whether or not we would be able to make a system that was secure enough to handle transactions like that. Additionally, we felt that it would make our users feel much more comfortable to have a different way of handling transactions than what we build in. The only real benefit to this option was to have everything taken care of by us, but ultimately the risks outweigh the reward of this scenario. We heavily debated whether or not to include Venmo API calls to handle transactions. On one hand, it would provide users with an easy way to pay each other, and it would allow for more to be handled in the app. However, we also acknowledged that not everyone might want to make their payments through Venmo. Additionally, this would create problems when trying to send payment requests to users who don't have Venmo. We eventually decided that it would be best to not have to worry about transactions since that would create too many security concerns. We opted instead to allow users to find their own payment method, and

then have them mark expenses as paid for. We then added the idea to have a consolidation feature so that individuals could just keep track of everything that they owe each other so that they could settle all debts between two parties at once.

Non-Functional Issues

---

1. How will we host our backend and database?
    - Option 1; Render Hobby Tier
    - Option 2: AWS Lambda Free Tier
    - Option 3: Google Compute Engine Free Tier

Chosen Solution: Option 1

Discussion: When deciding on where to host our backend, a few key requirements came to mind: it must be reliable, allow for a straightforward path for the app to scale if the number of users grows significantly, and it must allow us to host the app for free without the risk of extra charges. Debating the first requirement, the group decided that using **P**latform **a**s **a** **S**ervice (PaaS) would allow us to focus on deploying our backend code without worrying about the underlying infrastructure (which can cause issues if mismanaged). Thus, since the Google Compute Engine is **I**nfrastructure **a**s **a** **S**ervice (LaaS), we decided to not move forward with that option. Continuing to debate reliability, the AWS Lambda Free Tier seemed to have the slight edge given that it comes from a larger company; however, the Render Hobby Tier servers also seem to have reviews that praise its reliability. Moving onto the second requirement, both AWS and Render provide straightforward paths where a simple upgrade in the tier purchased expands the services resources without needing to migrate anything. When researching the final requirement, we discovered that AWS Lambda is a “serverless” computing service, which means that the service automatically scales when the application requires more power. This can be a problem if the app, by accident, exceeds the limit and charges the team. Render, on the other hand, will simply pause the app from functioning, which is a safer option in our case. Thus, we have decided to use the Render Hobby Tier to host our backend.

2. How will we host our frontend?
    - Option 1: In the same Render Hobby Tier Server that our backend and database is on
    - Option 2: Github Pages

Chosen Solution: Github Pages

Discussion: Since we are attempting to (in this initial stage of development) reduce the amount spent on hosting services while keeping our app uptime as large as possible, it seemed sensible to host the frontend on a separate free hosting service in order to work under the bandwidth/project size limits that they place upon us. Since GitHub Pages is only able to host the front end portion of websites, it was the perfect choice to reduce the chance of hitting our limits. Also, hosting on a separate service allows us to notify users if the Render Hobby Tier goes down for whatever reason, without rendering the entire web app useless.

3. How will we update room state so that it remains current?
    - Option 1: repeatedly check the time and state of the room while the user is logged in so that the room state updates immediately and remains current.
    - Option 2: check the time when the user opens the room, and then check every 5 minutes to update the status of a room
    - Option 3: only check upon the user opening a room.

Chosen solution: Option 2

Discussion: A big concern for us when discussing this subject was the amount of work that we asked of the server. We didn't want to clog the server with traffic that wasn't important, but simultaneously we want our displays to remain current so that our users can have a dynamically changing display. As a result, we opted against option 3 because it would never update the display for the user even if the status of the room changed. However, we didn't want to repeatedly ask the server to send the general information about the room since that would create a lot of traffic. As a result, we opted to only check for updates every 5 minutes, or when the user makes an update to the room. That way, initial changes would be displayed, and users would see the changes that their group members make within 5 minutes even if they don't update anything.

4. How can we store and access the roommate groups efficiently without utilizing an excessive amount of storage/computational steps?
    - Option 1: Each user JSON entry in the database has a list of codes that hash to where the roommate groups are in the MongoDB database in order to read the details for this room.
    - Option 2: Each user JSON entry in the database contains all the information for each roommate group that they are in

- Option 3: Each user JSON entry in the database contains a list of the names of the roommate groups that they are in that is then checked with every group in the database until a match is found.

Chosen solution: Option 1

Discussion: After short discussion, we were able to disregard Option 3 because, while the simplest option, it is very inefficient to check against every single roommate group in existence. Looking at Option 2, we were also able to dismiss it because storing the information of every room would not only be incredibly inefficient space wise, but it would also create synchronization issues as, everytime a roommate makes a change to the group, every single copy of the room needs to be modified across the database. This would use more computational power than necessary and can slow response times. However, Option 1 provides a perfect balance where there is only one copy of the room within the database that can also be accessed using an O(1) operation, significantly reducing response times and computational resources used. Thus, we have selected Option 1.

5. How will we handle collecting the information for the master room?
    - Option 1: each user is given their own master room object associated with them and all updates to a room will update each member's master room.
    - Option 2: when a user attempts to access their master room the server will send the files from each of their rooms and congregate the data

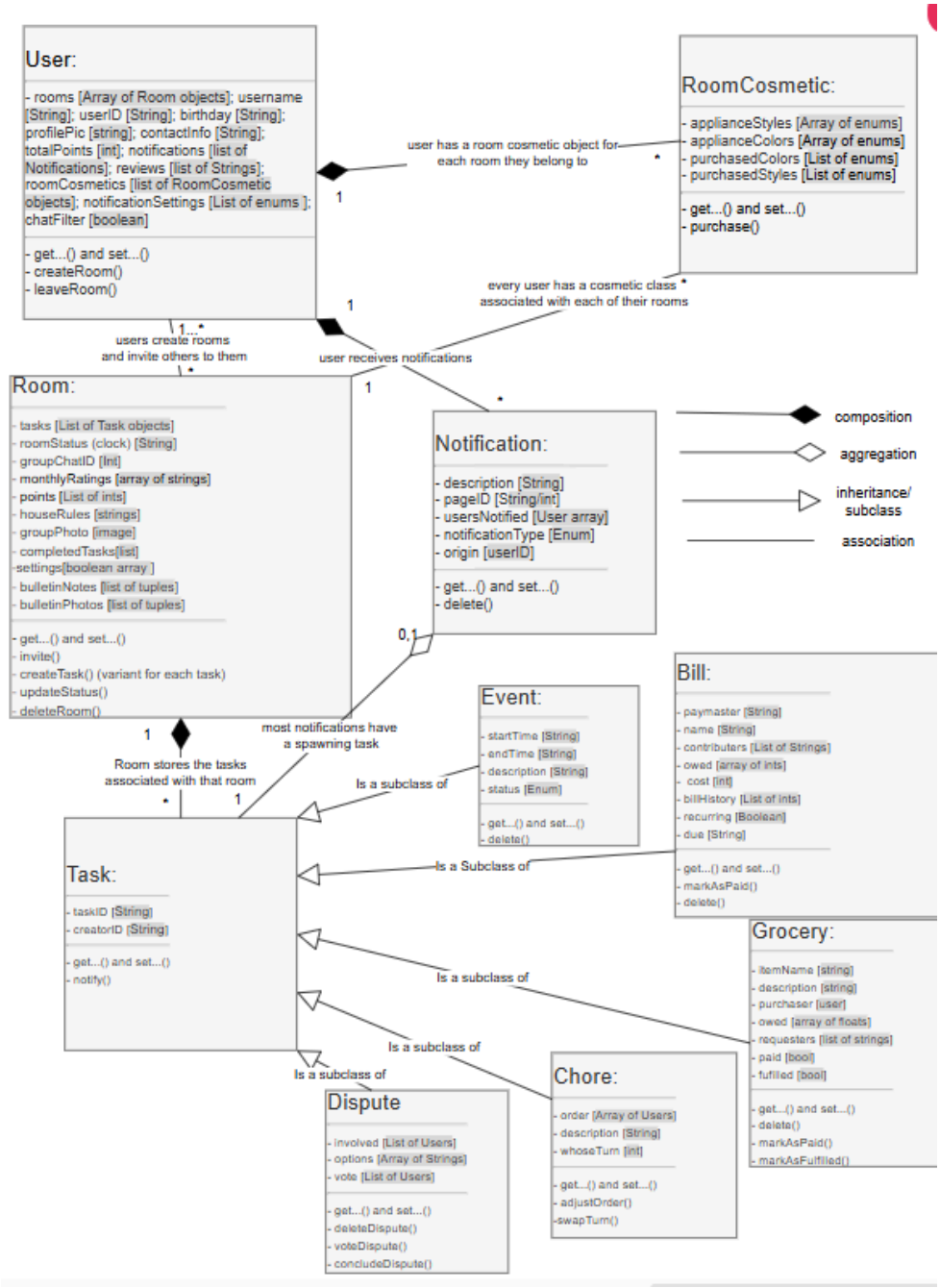
Chosen Solution: Option 2

Discussion: The group's big concern was limiting the amount of storage space we took up while balancing that with speed at runtime. We didn't want to leave the user waiting for long, especially since we believed that the master bedroom would be very useful. For that reason we were initially considering giving each user their own instance of a "master room" object, but then we realized that would mean everything stored in the database would also have to be stored in multiple locations since a single chore in one room would then have to be stored in that room and every member's master room. As a result, we would be quadrupling our storage space even on a room of just 3 people. Additionally, that would drag down the time of every operation since there would be more work required. For that reason we decided that we would rather pay the price of extra time only when a user is attempting to access their "master room". When a user accesses their "master room", we will simply compile all the information we need at that point in time. A

large part of the reason for this decision is because we realized that most users won't be involved in more than a few rooms, so we estimate that choosing to access each room as opposed to having a dedicated "master room" will simply multiply the time for this one operation by the number of rooms that the user is in, which we estimate will be low. However, if we instead chose to store a dedicated "master room" for each individual, then we would be multiplying the time for most operations by the number of people in that room, which can be very high (imagine a frat or a group sharing a house). So we picked option 2 as we believe it will be most space and time efficient.

Design Details

Class Design



The association between user and room is that a user can be a part of many rooms, but also each room can have many users. That's what is meant by the * to 1…* relationship.

Descriptions of Classes and Interaction Between Classes

Class descriptions and fields

The following are the classes that we have in our program: User, Room Cosmetics, Room, Notification, Tasks, Events, Bills, Disputes, Chores, and Groceries. Here are their descriptions and fields:

- **User:** Represents a single user of the program. It will store the user's ID, username, birthday, profile picture, contact info, email, notifications, reviews, notification settings, chat filter settings, and the rooms that the user is in
    - userID: A unique number for identifying users in the database, the current plan is to use a Google ID assuming we can get that from the API, otherwise, we will simply create our own
    - username: A string set by the user to display their name
    - birthday: the date of birth of the user
    - profilePic: a photo the user can upload and have displayed as their icon
    - contactInfo: text field where users can add information on how to contact them
    - notifications: a list of notifications that the user has received
    - reviews: list of strings that contain ratings from their previous roommates so that users may know where they may be doing well or need to improve
    - notificationsSettings: list of enums that indicate what kind of notifications that the user wants to receive (ex. User.notificationSettings = {CHORES, BILLS, NO_CHAT})
    - chatFilter: boolean allowing the user to always filter profanity out of chat even if the room settings don't
    - rooms: list of each room that this user belongs to
    - totalPoints: sum of all points accrued from all rooms (used to decorate master room)
- **RoomCosmetic:** The room cosmetic class contains the information for a room's appearance for a specific user. A user will have a room cosmetic class corresponding to each room they are in. The room cosmetic class will contain the colors for each appliance, the style of each appliance, the purchased colors, and the purchased styles.
    - applianceColors: An array denoting what color is set for each appliance
    - applianceStyles: array denoting what style is set for each appliance

    - purchasedColors: array denoting which colors are purchased for each item
    - purchasedStyles: array denoting which styles are purchased for each item
- **Room:** Composed of a list of tasks for that room, the status of the room, a group chat ID, the points for each member of that room, the ratings for that room, which features are enabled for that room, the rules for that room, a group photo, a list of tasks completed, and the settings for the room.
    - tasks: A list of tasks for the room, each of which is its own instance of the tasks class
    - roomStatus: the current status of the room based on the quiet hours
    - groupChatID: ID of group chat to hash to the correct chat in the database
    - points: list connecting each member to their points in that room
    - monthlyRatings: list of the past ratings that each member gave the room in the previous month
    - houseRules: a list of Strings that the group members will add and confirm into the room's rules
    - groupPhoto: a photo for the room to have on display.
    - completedTasks: a list denoting who has completed how many tasks of each type, this will later be used for the leaderboard tracker
    - settings: will be a boolean array denoting which features of the room are enabled/disabled for the room
    - bulletinNotes: List of tuples with the first element of each tuple corresponding to the note's coordinate within the notes section of the bulletin board, and the second element is the text content of the note
    - bulletinPhotos: List of tuples with the first element of each tuple corresponding to the note's coordinate within the photos section of the bulletin board, and the second element is the image id of the photo
- **Notification**: Notifications will be messages created when the user needs to take some form of action or needs to be made aware of something. A Notification will consist of a description/message, a pageID/path to reach the object that spawned the notification, a list of users notified, a type for the notification, and the ID of the user that spawned the notification
    - description: a string containing the message that the notification will display, this is essentially the body of the message
    - pageID: a means of linking the notification back to the page/object that spawned the notification.
    - usersNotified: a list of users who were notified by this notification

    - notificationType: an enum to describe what type of message (chat, task, dispute, etc)
    - origin: userID of the user who caused the notification, or a default ID for notifications spawned by the RoomEase platform
- **Task**: This is an abstract class for all tasks (events, chores, bills, disputes, groceries), it will contain a taskID, and an ID of the user who created the task
    - taskID: unique ID associated with the task so that it can be added, managed, and removed from the database
    - createrID: userID of the user who created the task so that it is known who created the task, and to control who can alter the task
- **Event**: Subclass of Tasks. Handles the scheduling of quiet hours and other events, will contain a starting date/time, end date/time, a description, and the status of the room at that time.
    - startTime: contains the starting time and date of the event (formatted like "HOUR:MINUTE MM/DD/YY")
    - endTime contains the end time and date of the event (formatted like "HOUR:MINUTE MM/DD/YY")
    - description: string input by the user when creating the event which will describe what the event is about
    - status: enum or const Strings to denote the status of the room at the time (ex: GUESTS_QUIET, NO_GUESTS_QUIET, NO_GUESTS_LOUD)
- **Bill**: subclass of tasks, will represent a bill that the room has to pay. Will contain information of who needs to pay, the paymaster, the name of the bill, the cost of the bill, the recurring status of the bill, the bill history, and the due date of the bill.
    - owed: float array the size of the roommate group that indicates how much each user owes for the bill [the paymaster will have their amount automatically set to zero] (ex. User 1, 2, 3, 4, and 5 are splitting a $50 Netflix subscription with User 1 as the Paymaster, so the array will look like [0, 10.00, 10.00, 10.00,])
    - paymaster: userID of the member who has been named the "paymaster" for that bill, they would be considered responsible for footing the bill upfront, and getting paid back by the other group members.
    - name: String containing the name of the bill
    - contributors: List of the userIDs of the users that are contributing to the bill
    - cost: the cost of the bill
    - recurring: boolean denoting whether or not the bill is recurring

- billHistory: list containing the previous costs of the bill if it is recurring, will be empty if the bill is not recurring
- due: the date when the bill is due (formatted like “MM/DD/YY”)

- **Dispute**: subclass of tasks. It will be used for handling disputes that arise amongst roommates. It will contain information on the people involved in the dispute, the options for what to vote on, and votes.
  - involved: list of users involved in the dispute, and their sides
  - options: list of strings describing which what the different options for resolution members can vote for
  - votes: list of who voted for which option
- **Chore**: subclass of tasks. The chores class will be used to handle the chores that the group has to do. It will contain a description, the order of turns, and whose turn it currently is.
  - description: string describing what the chore is/what the user is expected to be done.
  - order: array with the order in which members will be completing the chore
  - whoseTurn: whose turn it is to do the chore right now (index of the order array)
- **Grocery**: Groceries will also be a subclass of tasks, it will be specific to grocery items, or other items which roommates would like purchased. It will contain information on the item name, who paid for it, a description, the amount owed, whether or not it has been paid for, and whether or not it will be reused.
  - itemName: string containing the name of the item (ex. bananas)
  - purchaser: ID of the roommate who purchased the item
  - description: string describing the item/instructions (ex. 2lbs , only purchase if under $5)
  - owed: float array the size of the roommate group that indicates how much each user pays for the item (ex. User 1 and 4 want to split a $5 jug of milk, so the array will look like [2.50, 0, 0, 2.50])
  - requesters: List of the userIds of Users that have either requested the item or want to join in splitting the item (ex. User 1 requests eggs and adds User 2, 3, and 5 to the Requesters list for them to split)
  - fulfilled: boolean for if the item request has been fulfilled
  - paid: boolean for if the item has been fully paid for

Class methods

- **Universal:** the following methods apply to basically all classes. They are the get and set methods, many of the features will be implemented through a series of these get and set methods. Other methods were mostly retained for actions that require more than updating attributes of a class.
    - Set: sets an attribute of a class to a specific value
    - Get: retrieves the value of a specific attribute
- **User:** In addition to universal methods, a user has the ability to create a room, and leave a room, and there are API calls that will be involved with the login to this class.
    - createRoom: creates a new room and automatically includes the user as a member of the group. Will also create a new Room Cosmetic object for that user to go with that room.
    - leaveRoom: removes that user from the room, will also notify them of any outstanding debts to the room members, and will notify each room member of their departure. Will additionally remove the associated Room Cosmetic object.
- **Notification:** the notification class only has one unique method, which is to delete.
    - delete: removes this notification from the User's list, if the notification is still in other's notification lists, it will remain in those. If this user was the last to remove that notification, it will be removed entirely.
- **Task:** This only has one unique method, notify.
    - notify: when activated, this will create a notification object that will be sent to notify relevant users of task updates
- **Event:** the only unique method that the Event class has is the ability to delete itself when called
    - delete: when called, this will remove itself from the database and create and send a notification object indicating that the event is canceled (if the method is called before end time) or that the event has completed (if the method is called as a result of the end time passing).
- **Grocery:** This has a couple of unique methods that will mark itself as fulfilled, paid, and deleted
    - markAsFulfilled: This, when called, will set the "Purchaser" field to be whoever called this method, set the amount owed by all the users who requested the item to the price the purchaser entered divided by number of users involved (if the Purchaser also requested the item, their amount owed will remain at zero), and send a notification out to all of the users involved.
    - markAsPaid: This will set one of the user's "amount owed" value in the object to 0 and send the user a notification indicating that the Purchaser officially marked them as paid.

If this is the last user that needs to pay the purchaser, the function will then set the paid field to true

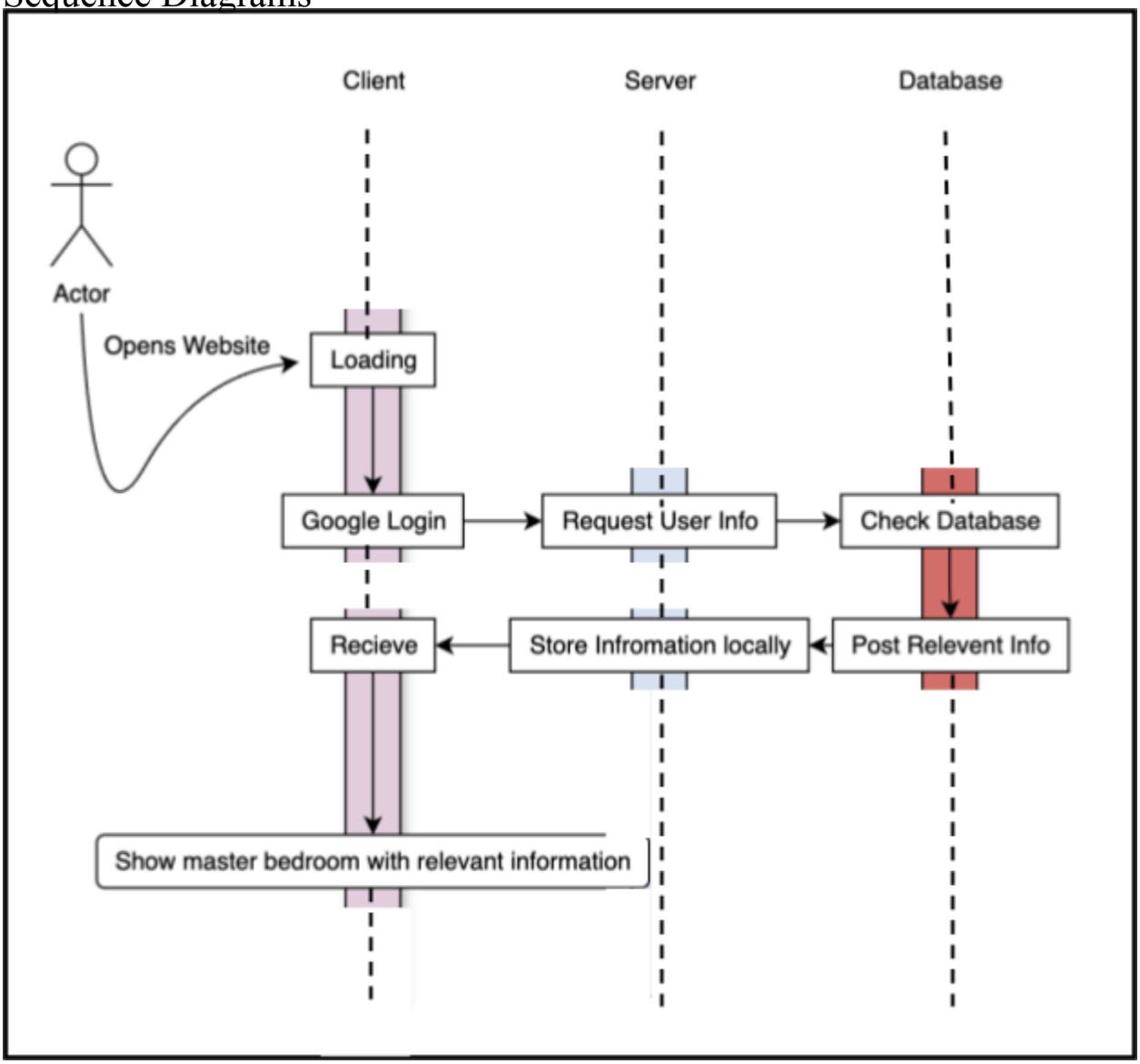
- delete: This will clear the object from all lists that it is within and remove the objects information on the database.

- **RoomCosmetic**: the only unique method that the room cosmetic class has is the ability for a user to purchase an item.
  - purchase: checks if the user has enough points in the corresponding room and that the item is currently unpurchased. If so, it will unlock the item for the user and alter their points accordingly.
- **Room:** the room class contains many unique features including the ability to invite others to a room, the ability to create tasks, check for updates to the room, and the ability to delete the room.
  - invite: will send a notification to another user asking them to join the room, upon accepting they will be automatically placed in the room, and a room cosmetic object will be created for them.
  - createTask: creates a task of the specific type (there will be multiple variants of this method all with fairly similar functionality). This task will then be added to the tasklist of the room and the status of the room will be automatically updated.
  - updateStatus: will check for any changes made to the room, and confirm the current state of the room with respect to quiet hours and other items.
  - deleteRoom: removes all tasks from this room and then removes from each user's lists and removes each user's room cosmetic classes. Will send each member a notification about the room being deleted and any outstanding debts to each other
- **Disputes:** the disputes class has three methods to vote on a dispute, delete a dispute, and conclude a dispute among the master room or a certain room. After concluding a dispute, the decision will be "posted" on the discussion board.
  - voteDispute: allows a user to vote on a specific dispute, and which side(s) they would like to take on the dispute.
  - deleteDispute: removes a dispute from the dispute queue, and restricts the user(s) who deleted the dispute from issuing another dispute for some set time. Deleted disputes will remove any decisions that were in it.
  - concludeDispute: Concludes a dispute after a given time frame. Time frames can be set in the master room settings, and disputes end by the decision which has the most votes.
- **Bills:** the bill class has a few unique features: delete, and mark as paid

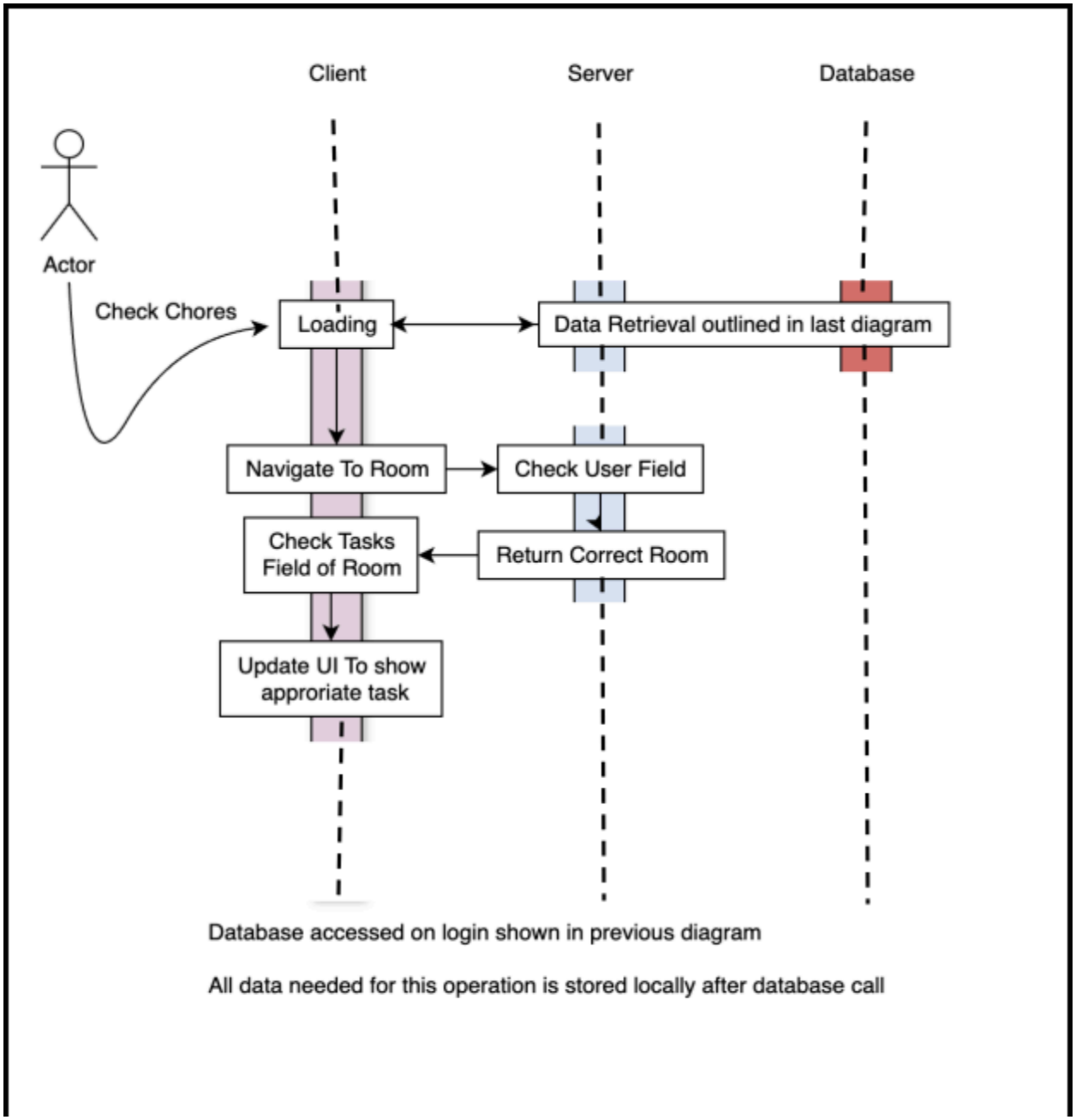
- delete: this will remove the bill from the tasklist of the room and remove the bill entirely, if there are still any outstanding payments, a notification will be sent
- markAsPaid: again, there will be multiple instances of this method, one will mark a single user as having paid, and another will be the whole group. It will clear all balances for this bill and update the fields accordingly. If the bill is not set to recurring, then it will be deleted when everyone is marked as paid.

- **Chore:** the chores class has two unique features including the ability to adjust the order in which users complete their chores and the ability to switch the turn to the next assigned user
  - adjustOrder: this method will allow the users to modify the order of the line in which users complete their assigned chores.
  - swapTurn: this method will update the variable for the index of the chores (whoseTurn) so that it will then be set to a specific user's turn. This method will be overloaded so that there is also a default to simply switch to the next in line.

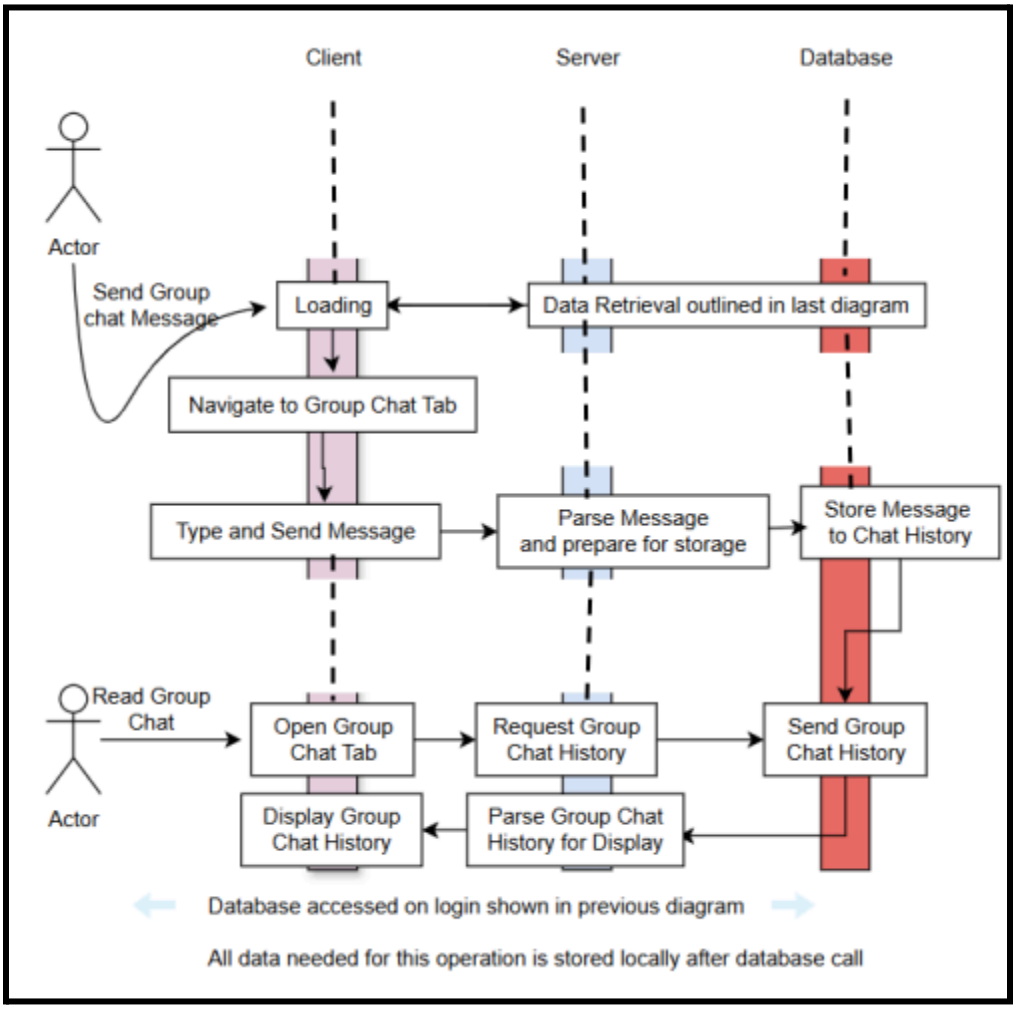
Sequence Diagrams



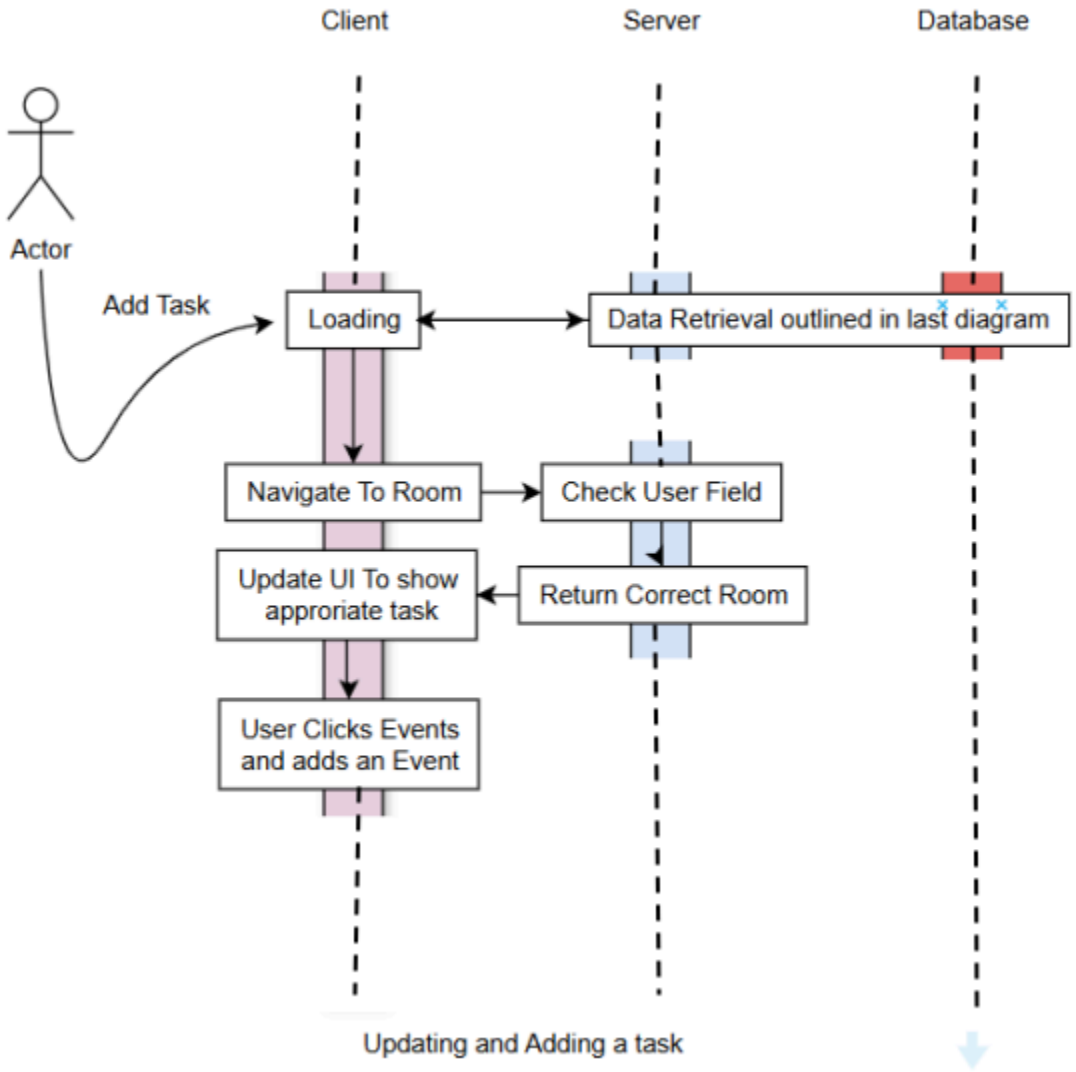
- User First Boots Up System



- User Checks Current Chores form task section



- User sends message to group chat that is viewed by another user



- User Adds a new event for the room

UI Mockups

Figure 1: Initial mockup for the mobile app

Above, you can see the original draft we had for our RoomEase app. Several aspects of the initial design will be continued into future versions of the design. When the user opens the app, this is sort of the room-esque view they will see, with some UI boxes for the kitchen, chores, grocery list, etc., and whatever else the user may want.

Figure 2: Second variant of the UI framework

As seen in figure 2, we have a better framework for a specific room within a master room. This figure includes both a fridge and a trash can. Selecting these objects within a web app will prompt a pop-up where you can see the relevant RoomEase features which we will implement in the web app. Every user will be able to customize their room, but this is just one example. This is then extended into .jsx files and react components, which can be seen in the figure below:

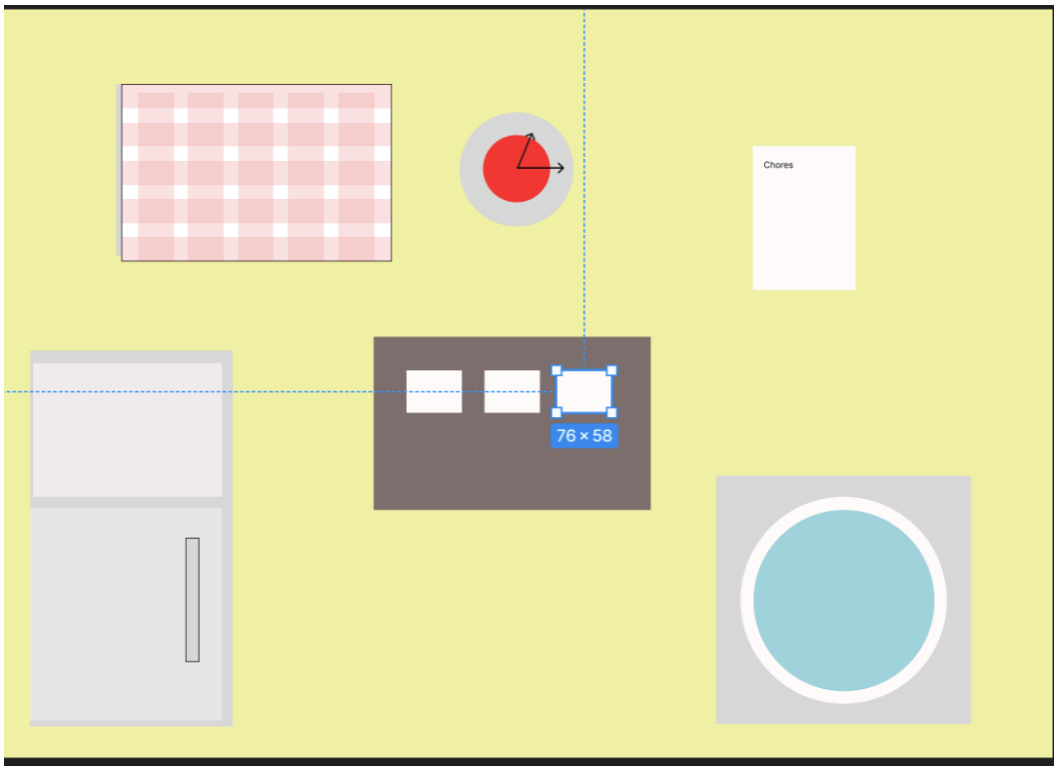


Figure 3: Third UI framework using Figma

In figure 3, the Figma prototyping started, and all of the related objects in the “room” had been structured. We plan to use this Figma project when creating our React.js components which will eventually fit the final version of the project.