RooMate

RooMate Architecture Document

By Ritvik Upadhyaya, Corey Pett, Aaron Levin, Zhanpeng Zeng, Hunter Koeshall, Seongjae Kim

***10-12-2016*, version 1.0**

Document Revision History

***10-12-2016*, version 1.0**

Table of Contents

1. System Architecture

1.1 Overview

RooMate will use the Model-View-Controller architecture. This architecture is practical for RooMate because the modular nature of MVC will make it easier to share one model amongst multiple devices. The model and the controller will be on the backend, and the view will be presented on the iOS front-end (and eventually Android too). The communication between the view and the controller occurs via HTTP/HTTPS protocols. All data will be processed and modified on the backend. Objects within the model will have different authentication tokens, so that when data is modified it can be verified on the server before changes are actually made.

1.2 Model-View-Controller Architecture

1.2.1 Model

The model will contain all user and group data, including which group each user belongs to, which items belong to each user, etc.. Having the model in the backend is crucial to the success of the project, as this is how we will synchronize data across multiple devices. Having the model closer to the controller (both in the backend) will also make processing substantially more efficient, because data can be processed in the backend as soon as the app is opened, as opposed to processing data locally after it is is retrieved.

1.2.2 Controller

The controller will reside both in the front end and the backend. The majority of the controller's implementation will be on the frontend, as that will be how the view communicates with the backend model. When the view is modified, the controller will handle the modifications and make appropriate HTTP requests, which will be handled by the controller on the backend to access/manipulate the model. Because part of the controller is in the backend, this will be where push notifications are processed.

Anticipated API's that will be used are as follows (but not limited to): AlamoFire for networking/HTTP requesting, SwiftyJSON for handling JSON objects, and Charts API (https://github.com/danielgindi/Charts) for handling data illustrations locally with Swift.

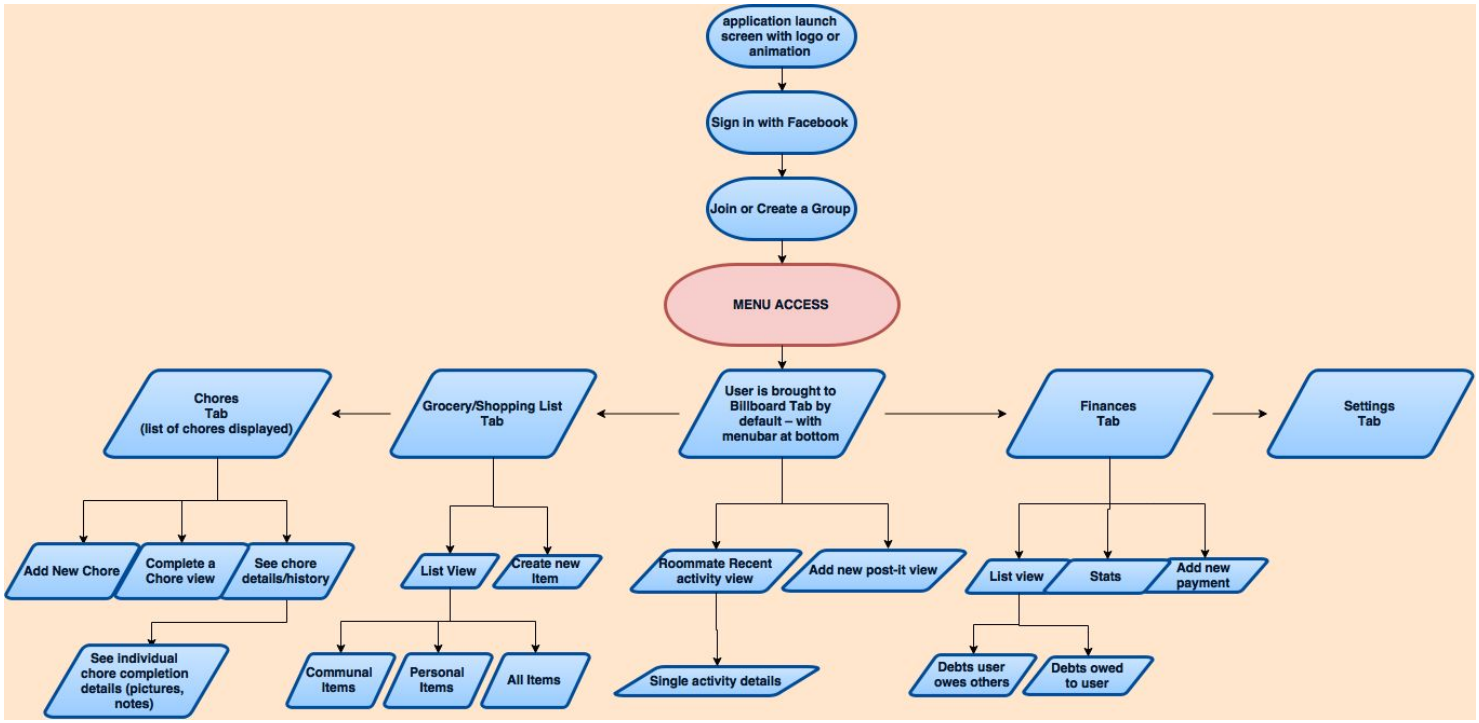
1.2.2 View

The view will be stored locally (on iOS and eventually Android) and will communicate with the model via the controller (both local and backend controllers). For the duration of this course, the iOS version will be the main focus of the project, and will be developed via Apple's Swift library on the Xcode IDE. The Xcode Interface Builder will also be heavily utilized to develop the user interface.

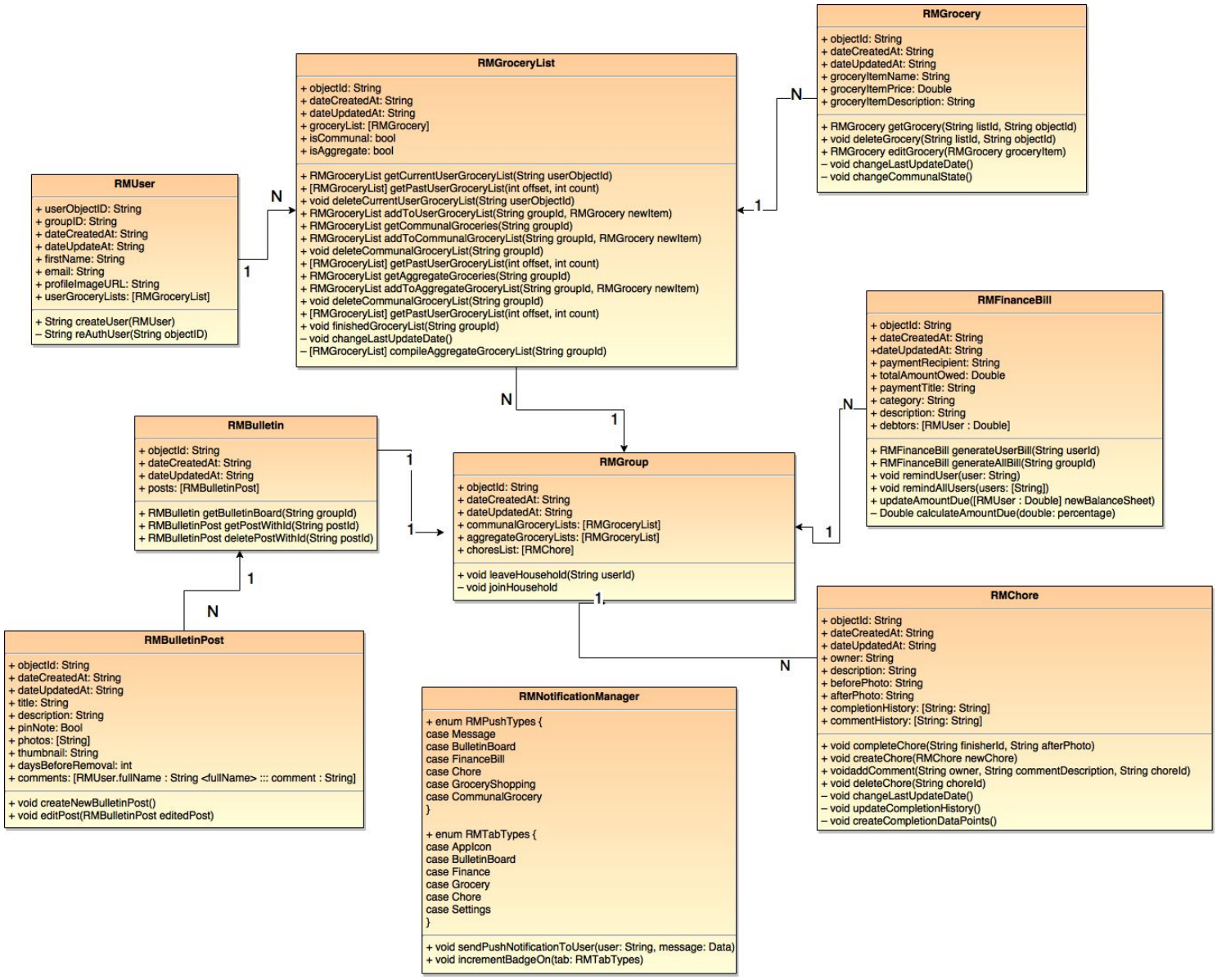
1.3 Notifications

Push notifications will be sent as JSON objects to Apple's Push Notification server from the backend controller, and then Apple will send the notifications to the devices, where the local controller will interpret the requests and adjust the view accordingly. The JSON objects contain relevant information to push to the users, which the controllers will be implemented to interpret. The server will know at what times to send JSON objects, and the objects will contain information relevant to the notification, such as which user to send the notifications to, what the message should say to that user, etc.

1.4 User Interface Architecture Hierarchy



1.5 API UML Diagram



1.6 Alternate Design Considerations

We considered using a Pipe and Filter architecture, but MVC seemed like a more logical solution for a multitude of reasons. Pipe and filter has potential to work, but it would complicate our communications between very distinct features of the application. It is less modular, which means it may be harder to debug and each function would be reliant on other functions. With MVC, this is not an issue, as all components of the app architecture are very distinguishable.

Also with MVC, the connections between our shared data source and local content is much more obvious (making debugging easier) and the code will be substantially more modifiable with an MVC architecture.

Event based systems are another approach that we considered. In reality, Xcode, iOS, and Swift already utilize this technology, and this will actually help us construct the app even more modularly.

We also considered using a layered approach, but realized MVC is more appropriate for consumer app development. The layered approach would be useful for the substantial abstraction we'll be using throughout the app, but the biggest fallback of the layered approach is that it will be exponentially harder to identify bugs, and it's harder to communicate with independent structures within the app (i.e. less modular). The decrease in modularity is a significant setback of this approach and we will absolutely not utilize this design philosophy while implementing our application.

2. Design Details

2.1 API Design

RMUser

This is the base user class of the project. All the user information will be stored within this object. The token, and the unique user ID will be stored here. Another property tied to the user will be the user's personal grocery list.

- **Properties:**
  - `userObjectId: String` // Also known as unique identifier
  - `groupId: String` // For what group they are in
  - `dateCreatedAt: String`
  - `dateUpdatedAt: String`
  - `firstName: String`
  - `lastName: String`
  - `email: String`
  - `profileImageUrl: String` // URL to user's facebook image, download it.
  - `userGroceryLists: [RMGroceryList]` //All the lists user has ever made
- **Public Functions:**
  - `String createUser(RMUser)`
    - String is the auth token
- **Private Functions:**
  - `String reAuthUser(String objectId)`
    - Re-issue an auth token for a user with Object ID. Will involve calling facebook for a new token and storing that token.

RMGroceryList

Grocery list is the superclass to the different types of lists we have: UserList, CommunalList, and AggregateList. This list is the data structure that will hold all the common properties of the list and the data of all the grocery items corresponding to the list. It will be responsible for populating the grocery tabs! The user can opt to edit, delete or view this list in detail.

- **Properties**
  - `objectId: String` //Unique identifier for each list
  - `dateCreatedAt: String`
  - `dateUpdatedAt: String`
  - `groceryList: [RMGrocery]`
  - `isCommunal: bool`
  - `isAggregate: bool`
- **Public Functions**
  - `RMGroceryList getCurrentUserGroceryList(String userObjectId)`

        - Get the most recent version of the user's personal grocery list
    - `[RMGroceryList] getPastUserGroceryList(int offset, int count)`
        - Get a particular set of older grocery lists of the user. Starting at the offset and getting as many old ones as specified.
    - `void deleteCurrentUserGroceryList(String userObjectId)`
        - delete the most recent or the current personal list of the user
    - `RMGroceryList addToUserGroceryList(String groupId, RMGrocery newItem)`
        - Add a new grocery Item to the given list

    - `RMGroceryList getCommunalGroceries(String groupId)`
        - Get the most recent version of the group's communal grocery list
    - `RMGroceryList addToCommunalGroceryList(String groupId, RMGrocery newItem)`
        - Add a new grocery Item to the given list
    - `void deleteCommunalGroceryList(String groupId)`
        - Delete the most recent or the current communal list of the group
    - `[RMGroceryList] getPastUserGroceryList(int offset, int count)`
        - Get a particular set of older communal grocery lists of the group. Starting at the offset and getting as many old ones as specified.

    - `RMGroceryList getAggregateGroceries(String groupId)`
        - Get the most recent version of the group's aggregate grocery list
    - `RMGroceryList addToAggregateGroceryList(String groupId, RMGrocery newItem)`
        - Add a new grocery Item to the given list
    - `void deleteCommunalGroceryList(String groupId)`
        - Delete the most recent or the current aggregate list of the group
    - `[RMGroceryList] getPastUserGroceryList(int offset, int count)`
        - Get a particular set of older aggregate grocery lists of the group. Starting at the offset and getting as many old ones as specified.
    - `void finishedGroceryList(String groupId)`
        - Marks the given grocery list as complete
- **Private Functions**
    - `void changeLastUpdateDate()`
        - Changes the last date a list was updated.
    - `[RMGroceryList] compileAggregateGroceryList(String groupId)`
        - Goes through all the private user's list and compiles it into one list.

RMGrocery

RMGrocery is the core object for all the grocery tasks in the app. This will hold all the details like the item name, description, who created the grocery item, etc. It will be a part of the RMGroceryList class as a member of the RMGroceryArray. The user can opt to edit, delete or view this item in detail.

- **Properties**
  - `objectId: String` // Also known as unique identifier
  - `dateCreatedAt: String`
  - `dateUpdatedAt: String`
  - `groceryItemName: String`
  - `groceryItemPrice: Double`
  - `groceryItemDescription: String`
- **Public Functions**
  - `RMGrocery getGrocery(String listId, String objectId)`
    - Get the RMGrocery object with the given objectId in the given list. This is called to get more details about a Grocery item.
  - `void deleteGrocery(String listId, String objectId)`
    - Delete the RMGrocery object with the given objectId in the given list.
  - `RMGrocery editGrocery(RMGrocery groceryItem)`
    - Edit the RMGrocery object with the given objectId in the given grocery item.
- **Private Functions**
  - `void changeLastUpdateDate()`
    - Changes the last date a list was updated.
  - `void changeCommunalState()`
    - Has to find the grocery item and remove it from RMUserGroceryList and move it to RMCommunalList and assign a new groceryObjectId.

RMFinanceBill

Each RMFinanceBill object contains information relevant to making personal or group payments. The RMFinanceBill object stores a user who is destined to receive payment from other users, and a dictionary, where each person that owes money is a key and the amount they owe is the value.

- **Properties**
  - `objectId: String` // Also known as unique identifier
  - `dateCreatedAt: String`
  - `dateUpdatedAt: String`
  - `paymentRecipient: String`
  - `totalAmountOwed: Double`
  - `paymentTitle: String`
  - `category: String`
  - `description: String`

    - `debtors: [RMUser : Double] // <User: AmountOwed>`
- **Public Functions**
    - `RMFinanceBill generateUserBill(String userId)`
        - Generates the bill for a specified user based on the dues unsettled with other users in the group.
    - `RMFinanceBill generateAllBill(String groupId)`
        - Generates the bill for all users in a group based on the dues unsettled with other users in the group.
    - `void remindUser(user: String)`
        - Send a push notification reminder to a user asking for money to settle unpaid dues.
    - `void remindAllUsers(users: [String])`
        - Send a push notification reminder to all users asking for money to settle unpaid dues.
    - `void updateAmountDue([RMUser : Double] newBalanceSheet)`
        - After an amount has been paid through venmo, update the balance sheet for the group.
- **Private Functions**
    - `Double calculateAmountDue(double: percentage)`
        - Then use total from a given bill to add to the amounts due for the users in a group.

RMChore

RMChore is the basic class that encapsulates the details of the chores that a person has to do. It contains the information about who created the chore, the history of the chore, and all the comments made by roommates. This object will be used to compile the graphs and data required to show completion histories and other useful information pertaining to the chore.

- **Properties**
    - `objectId: String` // Also known as unique identifier
    - `dateCreatedAt: String`
    - `dateUpdatedAt: String`
    - `owner: String` // Who did the chore
    - `description: String` // AKA additional notes
    - `beforePhoto: String` // Base64
    - `afterPhoto: String`
    - `completionHistory: [String: String]` // <username : Time of completion>
    - `commentHistory: [String: String]` // <username: comment>

- **Public Functions**
    - `void completeChore(String finisherId, String afterPhoto)`

        - Mark the given chore with Id as complete, and update the completed photo for the chore.
    - `void createChore(RMChore newChore)`
        - Creates a new chore in the group.
    - `void addComment(String owner, String commentDescription, String choreId)`
        - Add a new comment to the specified chore.
    - `void deleteChore(String choreId)`
        - Delete the specified chore with the given choreId

- **Private Functions**
    - `void changeLastUpdateDate()`
        - Changes the last date a chore was updated.
    - `void updateCompletionHistory()`
        - Changes the last date a chore was completed.
    - `void createCompletionDataPoints()`
        - Changes the last date a list was updated.

RMGroup

The RMGroup object is the basic data structure that stores all RMUsers within a household.

- **Properties**
    - `objectId: String` // Also known as unique identifier
    - `dateCreatedAt: String`
    - `dateUpdatedAt: String`
    - `communalGroceryLists: [RMGroceryList]`
    - `aggregateGroceryLists: [RMGroceryList]`
    - `choresList: [RMChore]`
- **Public Functions**
    - `void leaveHousehold(String userId)`
        - Add a user to the house
- **Private Functions**
    - `void joinHousehold()`
        - Remove a user from the house (can only be done internally)

RMNotifcationManager

This object will handle all the push or local notifications sent to the app. It will be used by the backend to create a push notification. On the local device, it will be used to determine what kind of notifications did the device receive and how to parse that notification payload.

Enums

- `enum RMPushTypes {`

```
            case Message  // message or billboard
            case Billboard  // billboard item added
            case FinanceBill  // CurrentUser assigned a chore
            case Chore  // somebody did a chore
            case GroceryShopping  // somebody is going shopping
            case CommunalGrocery  // item added
        }
```

- `enum RMTabTypes {`

```
            case AppIcon
            case Billboard
            case Finance
            case Grocery
            case Chore
            case Settings
        }
```

Public Functions

- `void sendPushNotificationToUser(user: String, message: Data)`
  - Send a push notification to the given user with the specific message as data.
- `void incrementBadgeOn(tab: RMTabTypes)`
  - Update the count of unseen notifications on the specified RMTabTypes

RMBulletin

The RMBulletin object is the data structure that contains all RMBulletinPost objects and organizes them in the view, according to dateCreated, dateUpdated, and whether or not each bulletin post is pinned.

- **Properties**
  - `objectId: String` // Also known as unique identifier
  - `dateCreatedAt: String`
  - `dateUpdatedAt: String`
  - `posts: [RMBulletinPost]`
- **Public Functions**
  - `RMBulletin getBulletinBoard(String groupId)`
    - Get the bulletin board for the group
  - `RMBulletinPost getPostWithId(String postId)`
    - Get the bulletin board post with the specified Id for the group
  - `RMBulletinPost deletePostWithId(String postId)`
    - Delete the bulletin board post with the specified Id for the group

RMBulletinPost

This object contains all information relevant to a single post on the bulletin board. Each post's information consists of a title, description, pictures, thumbnail, and user comments.

- **Properties**
  - `objectId: String` // Also known as unique identifier
  - `dateCreatedAt: String`
  - `dateUpdatedAt: String`
  - `title: String`
  - `description: String`
  - `pinNote: Bool`
    - Determines whether the post is pinned (true) or not pinned (false)
  - `photos: [String] //Base64`
    - An array of the photos to be included in the bulletin post
  - `Thumbnail: String //Base64`
    - The image to display when the post is minimized
  - `daysBeforeRemoval: int`
    - Determines how long the post should be visible before it disappears
  - `comments: Dictionary // RMUser.fullName : String <Full name : Comment>`
    - Maps each user who commented to the content of their message
- **Public Functions**
  - `void createNewBulletinPost()`
    - Creates a new post on the bulletin board
  - `void editPost(RMBulletinPost editedPost)`
    - Updates a post, given input parameter that is the new bulletin-post object

2.2 Push Notifications

Apple provides the push notification services for all the apps on the app store. So we will have to use apple's service for that. According to apple's website:

> For each notification, the provider:
> -Generates a notification payload
> -Attaches the payload and a device identifier—the device token—to an HTTP/2 request
> -Sends the request to APNs over a persistent and secure channel that uses the HTTP/2 network protocol
> -On receiving the HTTP/2 request, APNs delivers the notification payload to your app on the user's device.

Generating Notifications:

The push notification payload will originate at the NodeJS server we have. We will package the content (4KB or less in size) and request the Apple's server to perform the push notification. Here, we have to be really careful of the content we package and make sure that it is readable by the device. To do that, we will take the help of enums defined in RMNotificationManager.

Local notifications however, are generated by the Node server and sent directly to the phone. This is done at every background fetch. When the phone fetches new data, it has a buffer time to calculate all the changes, and decide on the way to present these changes. We will take the help of enums defined in RMNotificationManager to achieve this.

Receiving Push Notification:

The push notification will be received by the device after subscribing to the push notification service. The RMNotificationManager class will classify what kind of data is encapsulated in the push notification. Then we will unpack the data as objects and properly display the data on the respective view.

Risks with Notifications:

There are a few risks attached to the push notifications. First we have to absolutely sure that the payload matches the specs defined in Apple's webpage. Next, the package has to be within the specified limit. If not, the entire push will be rejected. We also need to maintain a unique device ID for each of the devices subscribed for push notification. This list has to be completely separate from the rest of the database, because any corruption of data will render the push service disabled.

On the device's side of things, mapping types to objects would be very important. An error in the unpackaging will cause all the new data to be lost, and unrecoverable till the next fetch for that particular data type. The other risk is the security between the requests. All the notifications, local or push require a very secure exchange, so should be authenticated with the account tokens to prevent middle man attack.

3. Implementation Plan

3.1 Depedencies:

We have divided our project members into separate teams for work on separate components. These components include:

- UI Interface
- Server (Network)
- Server (Database)

By splitting the project responsibility into three separate components, each team will know exactly what they need to implement and will have accountability in their role. The development of the UI Interface screens can be worked on in tandem with the building of the backend by using mock data not provided by the network. The network server team will simultaneously be preparing the communication avenue (nodeJS) between the front-end and back-end. Finally, the network database teams will be creating the data storage and ensuring seamless data accesses and modifications. Below we list the implementation plan for each of the three teams through our project iterations.

3.2 Iteration 1:

During the iteration 1, skeletons of view, individual units, and server are implemented independently. Unit Testing will be done at the middle and end of the iteration to make sure each unit is implemented properly. The server and the mobile devices should be able to create new objects from the design locally. The only service that the user will be able to access with complete backend support will be the chores tab by the end of this iteration.

Key:

Local Models	Backend	Frontend

Task	Description	Time Units	Assigned To
Class Implementation	Create the RMUser, and the RMGroup on classes on the NodeJS Server	1	Hunter
Class Implementation	Create the RMGroceryList, and the RMGrocery classes on the NodeJS Server	1	Hunter
Chore Data	Ability to store and query chore data in database from the UI	2	Hunter/Jae
Class Implementation	Create the RMFinanceBill class on the NodeJS Server	1	Jae
Class Implementation	Create the RMBulletin, and the RMBulletinPost on NodeJS Server	1	Jae
Class Implementation	Create the RMUser, the RMGroup, the RMGrocery,RMGroceryList, and RMFinanceBill classes using swift	3	Ritvik
Class Implementation	Create the RMBulletin, and the RMBulletinPost using swift	1	Ritvik
Storyboard	Build a functional UI skeleton of the chore tab, the tab bar, and navigations to detailed views related to chore.	4	Aaron/Corey/Pen
Storyboard	Build a functional UI skeleton of the bulletin board	4	Aaron/Corey/Pen
Storyboard	Build a functional UI skeleton for a Facebook login screen	2	Aaron
Local storage using core data	Implement the local storage functionality for Bulletin board and chores using core data	5	Ritvik/Pen

3.3 Iteration 2:

In Iteration 2, all implemented units are integrated and connected to the database. Mostly, we are going to focus on Integration Testing to make sure that all units interacts with each other as planned. By the end of this iteration, System Testing would be started. We will make sure that the other services like the finance bill, and grocery completely supported by both frontend and the backend. This includes the interaction with other members in the household.

Task	Description	Time Units	Assigned To
Grocery Data	Ability to store and query grocery data in database from the UI	2	Jae
Finance Data	Ability to store and query finance data in database from the UI	2	Hunter
Bulletin Board Data	Ability to store and query finance data in database from the UI	2	Hunter
Unit Testing	Verification of the chores tab in every aspect.	1	Ritvik
Unit Testing	Verification of the billboard tab in every aspect.	1	Ritvik
Unit Testing	Verification of the login page in every aspect.	1	Ritivk
iOS Network Calls for Grocery	Design the iOS side of connection for the grocery tabs	2	Ritvik
iOS Network Calls for Finance Tab	Design the iOS side of connection for the finance tabs	2	Ritvik
Storyboard	Build a functional UI skeleton of the finance tab, the tab bar, and navigations to detailed views related to chore.	4	Aaron/Corey/Pen
Storyboard	Build a functional UI skeleton of the bulletin board	4	Aaron/Corey/Pen

3.4 Iteration 3

In this iteration, we are mainly focusing on the testing. System Testing will be done in the first half of the iteration, and Optimization and Beta Testing on the second half.

Task	Description	Time Units	Assigned To
Push notification requests from server	The server will create a package and send the request to Apple for the push notification.	3	Hunter, Jae
Unpackaging Push notification	Handle the unpackaging of all the push notifications, and display that on the app.	5	Ritvik, Aaron, Corey, Pen
Unit Testing	Modify scripts running for unit testing to ensure all the calls are successful.	2	Aaron, Pen
Venmo Integration	Venmo integration to pay for an unpaid bill	3	Ritvik, Corey
Push Notification Testing	Create tests to ensure push notifications and local notifications are working as they should.	3	Ritvik, Corey
Automate a user run	Create a long test to simulate all the calls a user will go through. This will ensure that both the backend and the frontend are working as they should overall	5	All

4. Testing Plan

Throughout the development of the RooMate app, testing will play a vital role in validating previously written code to ensure it is implemented properly during the development phase. Tests to be done are Unit Testing, Integration Testing, System Testing, Optimization Testing, Regression Testing, and Beta Testing.

4.1 Unit Testing

Each component is developed independently and hence tested independently. The goal of this test is to minimize the size of each test and thus maximize the test details. This will help us to avoid the situation in which a bug in one combined code causing us to fix the whole code. Instead, it will be easy to determine where in the code the issue resides and allowing for quick, seamless debugging.

4.1.1 IOS Unit Test

Our front-end code will be developed in Swift 2.3 using Xcode; test code will also be written in same environment. Separate tests will be written for each tab menu interface, and more specific functions will be tested inside. For example, when launching our app a single unit test may be adding one chore to the households communal chore list. A complete passing of this test would be to check and see the chores list for that user's household was successfully updated.

4.1.2 Server Unit Test

Our server will be deployed through nodeJS using Heroku, a platform as a service (PaaS) that enables developers to build, run, and operate applications entirely in the cloud. This in tandem with the Heroku MongoDB extension will grant us necessary tools for unit testing our server. Some examples of the unit tests for our server include, but are not limited to: Sending/receiving app data to the server to be stored/accessed in the database and certifying and verifying server-side authentication for a single user.

4.2 Integration Testing

Once every unit test is passed completely, we will begin testing the integration of several components. Since many of the features interact with each other, tests will focus on the proper behavior after integration. The two major tests to be done are the IOS Tab Integration, and server integration.

4.2.1 IOS Tab Integration

We will be integrating all developed xcode modules together into one front-end structure. The integration testing will make sure that integrated units still function properly by testing all cases in which two or more separate units interact. Primarily, framework will be tested after all tab units are completed. For example, we will need to ensure integration of payment information from Grocery tab to Finance tab when splitting up a single grocery bill among multiple users.

4.2.2 Server Integration

The RooMate app will store and retrieve data from the server constantly, and it is important to test and ensure consistent behavior of our back-end. This test should check all cases where the app stores and retrieves data. This include the history of the completed chores, payment, and bulletin board posting. To verify success in these tests, our back-end team will ensure that the server/database can handle a sufficient amount of simultaneous traffic when users submit or access data.

4.3 System Testing

The purpose of this test is to discover, and prevent, unwanted behavior of the system. RooMate will be tested in customer point of view using simulator, performing various user actions that generally occur. It will focus on unpreventable systematic problems such as tapping the screen during a middle of the one action, an unexpected signal loss, and no action performed in a long time. In addition, abnormal action involving data modification on the server should be checked to prevent faulty data.

Compatibility tests will also be needed to ensure functionality for various iPhone screen sizes currently available in order to confirm that the app fits on all cases.

4.4 Optimization Testing

We plan to optimize the CPU management so that the app does not stay on background and drag CPU usage. This test will check if there is any memory leakage, and also check which functions are running on background. By comparing the result to the list of the necessary background jobs, we will be able to modify our app to minimize the background running jobs and thus reserve CPU and battery.

4.5 Regression Testing

Regression testing is to prevent breaking the code when new features are added. After the integration testing is passed and all pieces are put together, we will perform Regression Testing development advances to make sure the system still performs correctly even after it is changed or interfaced with other softwares. For example, when adding the Venmo API to complete bill payments between two users, we must take preventative measures to ensure that any additional changes to the application will not affect the compatibility between RooMate and this vendor application.

4.6 Beta Testing

Beta version of our app is to be released to a few groups of testers after the last iteration. Since the app is only for IOS user, iPhone 6 or above is required as a tester. To generalize, the group will be selected with various number of participants, from two to five. The basic guideline and questions will be given to the testes, but they will also be free to use the app just as how they would normally and report the problem they might encounter. The questions include UI of each section, systematic error on various behavior, and customer recommendation. All feedbacks will be gathered and determined which problem is the most important to fix. Our client will join at this stage for better determination in customer point of view. We will then go through the list and fix each error.