RoomEase Software Design Specification

Alex Vrhel (avrhel), Sid Gorti (sgorti3) Jakob Sunde (jsunde), Omar Alsughayer (oasugher), Cheryl Wang (cwang7), Matthew Mans (mans1626), Weijia Dai (weijid)

System Architecture

Customer View

RoomEase is composed of several screens the user can quickly access from anywhere in the app. These screens correspond to the different features offered, which are listed as follows:

- Feed: this is the landing page of the application that displays when it is opened. The side menu and shortcut buttons at the bottom of the screen allow for quick and easy navigation between the areas of the app. A feed of important current information is displayed on this page, telling the user if any foods have expired, upcoming reservations.
- Fridge: keeps track of all of the items in your fridge, making note of their expiration dates and whether or not an item is shared between roommates.
- Lists: a shared note taking area where roommates can keep track of to-do lists, grocery lists, house wishlist, etc.
- Reservations: shared resources can be reserved from this page, allowing roommates to claim the use of the laundry machine, shower, tv, or other items added by the group.

All of these pages except for the feed view allow the user to add an item in the fewest presses possible.

Developer View

The main view module simply contains the menu bar including the slide out side menu. It then calls on the different page views to render themselves, injecting HTML and CSS into the main html document. This frees us from having to load javascript and CSS documents every time we change contexts since they are loaded upon start up and the main page is never left. It also allows for smoother transitions between pages.

When the user first logs in, a JSON object is submitted to the database storing the user's information, their name and their facebook userID. If they would like to create a group then their group information is entered into the database, storing their group name and groupID which will be generated by the database itself.

Whenever a user adds an item to any of the pages, it will be added to the respective module's storage and it will be added to the database. If there is no network connection, all items will be added to the database at a later time using PouchDB's functionality for adding data to a database. All of the data will be stored in lists, and each list will be sorted differently based on their type. The fridge items will be sorted in order of expiration date, lists from the note-taking section will be sorted by the date/time they were added, and all other items will be sorted based on the date of their occurrence.

Design Decisions

We initially planned on having separate classes & containers for each item (i.e. a fridge container that stores only fridge type items, or a reservations container for reservations items), but after some group discussion we thought that the type of the item/container would be the only differentiation between them. Thus, before beginning our coding we decided to

condense all of our item types into one item class that differentiates between items via a type field, and do a similar simplification with the type containers becoming a more generalized client storage. While coding, however, we discovered that our modules had more type-specific behavior than we had originally predicted, and so went back to a separated version of one controller for each page.

Our request handler was originally on server side, and it was going to be the "middle man," processing requests from the client and routing them to the correct database. However, we decided it would be easier to implement this functionality client side. Then we can take advantage of PouchDB's offline syncing capabilities, which is a very important aspect of our application. The request handler is now going to identify the type of item being added to or requested from the database in order to do the routing to the corresponding database, allowing us to have this module client side instead of server side. It is also easier to program in JSON client side.

We assume that we will be creating a slick enough UI such that roommates will find the app easy to integrate into their lives, as well as helpful overall.

Database Design

For our database, we chose to use CouchDB, a NoSQL database that stores entries as JSON objects. Every JSON object in the database will automatically assigned a unique id and a unique revision number. These values are auto-generated by CouchDB . JSON objects within the database will be able to refer to other JSON objects using these values.

Within CouchDB, we will have separate databases that store different types of items. Our CouchDB server will consist of the following databases: Users, Groups, Reservations, Lists and Fridge Items. Each User JSON object stores a UserID which is generated when the user logs in for the first time, a name, and a group. Users will be identified by this UserID, rather than the automatically generated ID provided by CouchDB for the user JSON object. Every other JSON object will be identified by the automatically generated id provided by couchDB. Each Group JSON object will contain the UserID of each group member of the group, along with the IDS of each Reservation, List and Fridge JSON object that is associated with that group. Each of the Reservation, List, Fridge and Item JSON objects will contain information on the creator of the JSON object, the group it belongs to, and data that is specific to that particular entry (EX: Fridge entries will have an expiration date, owner, ETC).

Diagrams

Class Diagram: See RoomEase_class_diagram.pdf

Sequence Diagram: See RoomEase_Sequence_Diagram.pdf

Process

Risk Assessment

1. **Not having a server side application causes too much burden for client device**
   Likelihood of occurring: medium
   Impact if it occurs: high
   Evidence for estimates: For our current design, almost all our code will be implemented on the client, meaning the client will have to process database queries and responses. Also, we keep cached copies of some of the information on the client side, and too much cached information may become too much for the client to handle. The impact of this problem can be quite serious, for our potential customers may refuse to use our app due to sluggish performance on the client side.
   To reduce likelihood/impact: we will try to find a good balance between what users need to access from the database and what is cached. For example, instead of caching all the future reservations, we will probably limit our reservations to the following 3 days or so. Also, not querying for all information at once can spread the load of processing database queries.
   Plan for detecting problem: We will continue to look into how much local storage our app is using for cached queries, and we will monitor the amount of processing power the app uses when making a database query. We will perform these tests on real-world devices.
   Mitigation Plan: If we find out our app really leaves too much burden to client device, we may switch from using PouchDB to implementing a server side application to process DB queries. This may require us to adjust our design and utilize group dynamics as needed.

2. **Risk of Fridge management feature being tedious to use**
   Likelihood of occurring: high
   Impact if it occurs: medium
   Evidence for estimates: We have had concerns for this since we started doing project proposal. Also our TAs and classmates have also expressed similar concerns. It does not affect the functionality of our app, but it will eventually make our app not user friendly.
   To reduce likelihood/impact: We will try to simplify the process for our users to enter food items. For example, we can provide shortcut icons for the items a user frequently enters so the user doesn't have to type every time. Also, we can add autocomplete feature for the names of the food items so users do not have to re-enter information for items they have entered before.
   Plan for detecting problem: We will perform usability tests after the basic fridge module is done to see if there is a need to add more features that simplify the food item entering process.
   Mitigation Plan: If this problem happens in the end, we will try to add the features mentioned above to simplify the food item entering process.

3. **Risk of not being able to handle race conditions**
   Likelihood of occurring: medium

Impact if it occurs: medium
Evidence for estimates: Since our app relies on PouchDB to handle database interactions, and with the design we have right now we do not have a server application, it is possible that we find out we are not able to handle race conditions associated with modifying entries in the database. We have discussed many race conditions that can occur, but there may be cases we did not consider.
To reduce likelihood/impact: We will likely use the error-handling features that PouchDB and CouchDB provide, but we may need to come up with our own method for handling database race conditions.
Plan for detecting problem: We will design unit tests that force a race condition to occur.
Mitigation Plan: If race conditions become difficult to manage, we may have to come up with a means to make certain user actions synchronized, or figure out a way to queue certain user actions.

4. **Handling different types of requests in one class results in too much coupling**
   Likelihood of occurring: medium
   Impact if it occurs: low
   Evidence for estimates: Right now we are handling different kinds of requests in one class. We do it this way so we can reuse much of our code, but it seems to be against the philosophy modularity.
   To reduce likelihood/impact: We will try to make the methods in our RequestHandler class that handle different types of requests comparatively independent of each other.
   Plan for detecting problem: When we do unit-testing we will pay special attention to our Handler class and see if it causes any inconvenience. Also, feedback from the team will be important in seeing if this design is too difficult to work with.
   Mitigation Plan: If it turns out to be an issue, we can always factor the module into separate modules.

5. **Database schema ends up being inefficient**
   Likelihood of occurring: low
   Impact if it occurs: medium
   Evidence for estimates: The data for our application is stored in different CouchDB databases (The databases are UIDs, groups, reservation, list and fridge items). To retrieve or post data we usually need to access more than one database, and each database query corresponds to one HTTP query, and too many queries could become highly inefficient.
   To reduce likelihood/impact: If our schema becomes too inefficient, we will modify our schema to use fewer databases for the information we have, and therefore fewer database queries.
   Plan for detecting problem: We will test the efficiency of our app frequently so we have a general idea of if our database design works well
   Mitigation Plan: We may need to combine the databases we have into larger databases. This requires us to update our milestones and utilize group dynamics as needed.

Schedule and Team Structure

The team is split into a backend and frontend group, with frontend including Cheryl, Jakob, Sidd, and Alex, and Backend including Omar, Matt, and Corie. Each of our members also move around depending

on what is needed. Each of our milestones (bolded in the schedule below) has a set of tasks for each team to complete in order to reach it. We plan on splitting up the tasks more specifically between each subteam as we get closer to the date of the tasks, based on availability of team members and what they want to work on. We communicate via Slack, a chat room application for teams, and Facebook, and we have weekly meetings on Tuesdays at 12:30 and Thursdays at 3:30, both in the Research Commons on campus.

Milestones and Tasks	**Done by**	**Time Estimate**	**Owner**	**Dependencies**
Code fully specified, in repository	**Feb 4th**		**All**	**All tasks listed directly below**
- Read up on PouchDB documentation	Feb 1st	~1 hr/member	Backend	
- Read style guides	Feb 1st	~1 hr/member	All	
- Decide on complete database schema	Feb 2nd	2 days (by Feb 3rd	Backend	
- Read up on HandlebarJS documentation	Feb 1st	~1 hr/member	Frontend	
- Read up on MaterializeCSS documentation	Feb 2nd	~1 hr/member	Frontend	
- Write all classes/modules/functions as stubs	Feb 3rd	~ 1 day	All	Understanding above documentation, system design finalized
- Comment on all classes/modules/functions	Feb 4th	~ 1 day	All	
Implemented get, put, into DB and local storage	**Feb 7th**		**Backend**	**All tasks listed directly below**
- Implement items, set up local storage	Feb 6th	1 person-day	Backend (Omar)	
- Implement get and put methods of request handler	Feb 6th	2 person-days	Backend (Matt, Corie)	
- Implement way to trigger get or put via UI	Feb 7th	2 person-days	Frontend	
- Implement unit tests for request handler get, put	Feb 7th	1 person-day	Backend	
Implemented list view, router to switch views	**Feb 7th**		**Frontend**	**All indented tasks listed**
- Implement list view (rough UI)	Feb 5th	2 person-days	Frontend	
- Integrate local data to show up on list view	Feb 6th	2 person-days	Frontend	mocked local data
- Implement one other view	Feb 6th	1 person-day	Frontend	

- Add routing logic to switch between views	Feb 7th	2 person-days	Frontend	
First usable prototype (just list module)	**Feb 11th**		**All**	**Request handler works for add/remove, list view implemented**
- Begin unit testing untested modules		2 person-days	All	
All views (not all interactivity or animations)	**Feb 14th**		**Frontend**	
- Implement fridge view	Feb 14th	6 person-days	Frontend	
- Implement reservation view	Feb 12th	3 person-days	Frontend	
- Implement feed view & interactions	Feb 12th	3 person-days	Frontend	
- Begin work on buttons			Frontend	familiar with materializeCSS
All request handler and syncing locally works	**Feb 14th**			
- Decide on and implement syncing scheme	Feb 14th	6 person-days	Backend (Omar, Corie)	
- Implement feed functionality	Feb 13th	3 person-days	Backend	
- Unit test all request handler methods	Feb 13th	2 person-days	Backend (Matt)	
- Integration testing with smaller database	Feb 13th	3 person-days	Backend (Matt, Corie)	
- Finalize local storage for list and reservation items	Feb 14th	1 person-day	Backend (Omar)	
2nd prototype (all views, list and reservation functional)	**Feb 15th**		**All**	Met above goals on schedule, list and reservation fully functional back to front
Packaged for Beta Release (3rd prototype)	**Feb 19th**		**All**	
- Finished fridge UI interaction and animations	Feb 17th	4 person-days	Frontend (Jakob, Sidd)	
- Error handling for race conditions in DB	Feb 17th	4 person-days	Backend (Corie, Matt)	
- All UI widgets for views integrated	Feb 18th	5 person-days	Frontend (Jakob,	

			Alex)	
- Frontend integration testing and unit testing	Feb 19th	4 person-days	Frontend (Cheryl, Sidd)	
- Implement FB login and authentication in backend	Feb 18th	6 person-days	Backend (Omar, Matt)	
- Implement rough login screen and group formation flow	Feb 18th	4 person-days	Frontend (Sidd, Alex)	
4th prototype	**Feb 23rd**			
- Finish up any unfinished main goals	Feb 22nd	8 person-days	All	
- Implement polish and animation	Feb 23rd	6 person-days	Frontend (Cheryl, Jakob)	
- Write user documentation	Feb 23rd	2 person-days	Alex	
Feature complete release	**Feb 26th**			
Add as many stretch goals as possible (timing uncertain)				
- Native push notifications		10 person-days		Work with phonegap libraries, sync functionality is robust and complete
- Who's home		8 person-days		
- Integrated payment		12 person-days		All views finished and have room to change to incorporate new UI elements

Test Plan

Backend Unit Testing:

For testing our backend, we will use Jasmine as our testing framework. We plan on testing whether or not we receive the correct values when a certain request is sent, whether a certain item is correctly added to the database, and so on. Mocking our Database will be fairly difficult and awkward to unit test, so we will be performing actual requests to our database when we perform our unit tests using tables in our database set specifically for testing. We will require that the person who wrote the code will be the person who writes the unit tests for it.

Frontend Unit Testing

To test our frontend, we will continue using Jasmine/Jasmine-JQuery. We will be using Jasmine primarily to check for correct display behavior of our UI elements, i.e. whether a popup displays block and

whether titles are correct. Once again, we will require that the person who wrote the code writes the tests for frontend.

System Testing

For System Testing, we will likely use a combination of the frameworks that we use for front end and back end unit testing. Much of what we will want to test at the Systems Testing phase will be “round trip” functionalities, or in other words, making sure that actions and events such as button presses will properly send out a request to the server and receive and display the proper information. We will explicitly state which tests are system tests and which tests are unit tests. We will write system tests as we complete modules and develop them to the point where they can interact with other modules.

Usability Testing

Usability testing will be performed in two separate ways. First, during early development, we will have everybody in the group report on any major issues regarding the usability of the application. Once our application is at a point where an average user could use the application, we will have Jakob and Alex along with their roommates, and Matt along with his roommates, test out the application. Any technical issues that occur will be listed in the bug tracking tool, and any issues that are concerned with UI layout and flow will be part of a separate Google Doc labeled “Feedback”.

Continuous Integration Testing

Our testing suite runs every night on the server containing our database. A simple script runs the tests and sends the results to the mailing list keeping everyone up to date on any failures allowing us to correct them as soon as possible.

Adequacy of Testing Strategy

Overall, this testing suite will be adequate when it comes to finding and addressing issues with our system, for it covers nearly all scenarios where bugs will manifest. Any issues that are not caught in automated testing will likely be found through our user testing.

Bug Tracking

We will use Google Sheets in order to tack all known bugs. For each bug, we will include the following: The date that the bug was discovered, who discovered the bug, a description of how to reproduce the bug, and a rating of how critical it is to fix the bug. If possible, a GIT revision may be given for when the bug first appeared. This will not be required.

Code Coverage Tool

We will use Blanket.js for our code coverage tool. We found that this tool provides the simplest means of testing for code coverage, and we found that this was the easiest code coverage tool to use with Jasmine.

Documentation Plan

RoomEase is going to include a HELP section that will guide new users through the app. Beside that, HELP will include a FAQ section and will link to the developer's website for a user to contact us.

Coding Style Guidelines

We will use the Google Javascript Style Guidelines and Google HTML/CSS Style Guidelines. We will enforce a code review process, where on each commit, the programmer is responsible for getting another member from their sub-team (frontend or backend) to review their code and approve it within a reasonable time frame (~½ a day).

Design Changes and Rationale (Beta Release, 2/19/2016)

The only changes we chose to make in the design were to use Jasmine for unit testing instead of QUnit because it was easier for us to set up in order to have the results mailed to the group. The other change we made was the way we would do continuous integration testing, we chose to have a cron job run a grunt script to run our suite of Jasmine tests. This was the easiest way we could have the results of unit tests sent to the group which allows everyone to stay up to date on the status of the repository and we would know of any issues at the very latest by the next day, if someone didn't run the suite of tests themselves before pushing their changes

Design Changes and Rationale (Feature Complete Release, 2/26/2016)

In our Testing section, we added an amendment that describes what Code Coverage tool we will be using. We found that this tool was the easiest tool to use with our testing framework. Also, we removed the documentation regarding chores and removed chores as a feature entirely. We decided to remove this feature for multiple reasons. First, we found that the scope of the features we wanted to implement was too large for our time constraints, so we had to decide on what features we wanted to cut. We found that the chores feature is something that other apps do and do better than we could do, and while it would be a nice to have, this feature does not make our app unique, while our other features do. Finally, if a user really wishes to create a chore schedule, they could use the Reservation/Scheduling tool to do so. While the scheduler is not meant for chores, a user could use it for scheduling chores if they really wanted to.

Design Changes and Rationale (Release Candidate, 3/4/2016)

When designing the app we believed that having one overall view, controller and local storage module would be the most efficient. However, we found while implementing the code that there was more type-specific logic involved in displaying, adding, deleting, and updating our items than we initially predicted, and that using the same controller for each page was messy and confusing to work with. Because of this, we ultimately went back to separating our views and controllers based on type. Each of the pages we have now has its own specialized controller that renders the page, stores its type's local copy of the database items, and holds other type-specific functions related to the UI. We have also decided to continue using Jasmine to test our front-end implementation.