Course Name: Software Engineering

*

Course Number and Section: **14:332:452:01**

Group #10

Drip ‘n Dash

Report 3

*

Website: https://sites.google.com/view/dashndrip/dash-n-drip

Group Members:

Elaina Heraty, Abdullah Bashir, Harshavardhana Dantuluri, HyunSik Kim, Karan Parab, Liam McCluskey, Nayaab Chogle, Nicolas Rubert, Roberto Cruz, Siddharth Manchiraju

Subgroup	Member	Contribution
1	Liam McCluskey	10%
2	HyunSik Kim	10%
3	Nicolas Rubert	10%
4	Roberto Cruz	10%
5	Nayaab Chogle	10%
6	Siddharth Manchiraju	10%
7	Elaina Heraty	10%
8	Harshavardhana Dantuluri	10%
9	Abdullah Bashir	10%
10	Karan Parab	10%

Table of Contents

Summary of Changes

- Added use cases:
    - Order cancellation for the Customer role at certain steps in a job.
    - Prefered Dasher - Customers can choose a Dasher to do their laundry instead of sending a request out for all Dashers to receive.
- Changed diagrams
- Discarded support for Speed Queen since they do not have any straightforward integration apk.
- Interaction Diagrams Updated
    - Pages/Activities in the app are shown in more detail, allowing more accuracy to what is in the code.
    - Customer Login and Dasher Login is combined into one diagram.
- System Sequence Diagram Updated
    - Accurately represents the timeline of events of our current build (Removed mention of quiz and Time Completion Estimation tracking as it is not implemented yet).
    - Blue to White gradient indicates direct interaction between the user and the database (Reverse direction applies as well).
- Removed Nonfunctional Requirement - The quiz we were going to implement in order to become a dasher has not been implemented so this seems redundant to keep in this report.
- Removed UI Requirement
    - No GPS functionality has been incorporated so the Customer would not be able to see how far the Dasher is from their dorm building.
- Traceability Matrix Updated
    - Added new use cases.

Customer Statement Requirement

Problem Statement

For most students, college life entails a significant amount of hard work, along with a healthy amount of time for extracurriculars. This allocation of a large portion of each student's day means that most students may not find the time to complete simple tasks, let alone their laundry. Laundry is a necessary chore for all students to perform, but with assignments, club events, and social gatherings scheduled almost every week, it is relatively hard to find the time. After all, not all students can return home and have their parents do their laundry for them. However, even if a student were to have time to do their own laundry, they may occasionally find themselves not feeling up to the challenge of partaking in a chore as tiresome as taking care of their laundry and would much rather have someone else do it for them. What may seem to be an everyday and simple chore, could turn out to be a problem for some individuals.

Customer Side Problems

In addition, not being able to do laundry can be detrimental to a student in a variety of ways. Having unwashed clothes can lead to a student falling ill, which would hinder their ability to succeed in school.With this in mind, having a helping hand every once in a while to assist with this task would be extremely beneficial. This method would make it possible for them to request and pay someone to do their laundry for them.

Additionally, there are students who do not know how to properly do laundry, potentially making this a daunting task in order to get clean clothes. On the contrary, doing laundry is a very intricate process that if not done properly, it could cause an enormous mess. There are many things to take into account when doing laundry, those of which include the amount of laundry detergent used, the temperature of the water, the type of material of the clothing article, and many other variables that an inexperienced person would overlook. It is through using the app that the same student would not have to worry about any repercussions they could bring about by not knowing how to do laundry.

However, the students who do have plenty of spare time could benefit from this service, turning this utility into a commodity for some extra money. Sometimes a student might just want someone else to do the laundry for them. There are times in which a person would rather do another activity with their free time- be it more studying, spending time with friends, or even relaxing- instead of having to do laundry. For this reason, these students would still be able to make full use of the app whenever they desired.

Dasher Side Problems

Although many students dislike doing laundry, there are students that don't mind it and would like to make some extra money by doing other peoples laundry. For these students, whom we call dashers, the main problems consist of needing an easy source of revenue, many part time jobs having a comparably high barrier of entry and non-flexible work hours.

The expenses of college students can add up quickly, leaving them in need of an easy source of revenue to fund all these costs. However, finding a convenient job on campus can be difficult, since some positions require a specialized set of skills, or cannot accommodate a student's busy class schedule. This is where Drip 'n Dash comes in; Unlike most other jobs that often require a long tedious interview process, Drip ‘n Dash only needs students to pass a quiz about how to properly do laundry to register as a dasher. After becoming a Dasher, that student is able to accept jobs in their own free time.

Suggestions from an Interested Customer

As a student in University, I do not have the time to do laundry with my intensive schedule. Although, I am unwilling to give my valuable clothing items to strangers to do my laundry for me. Services such as Facebook, Craigslist and TaskRabbit are high risk because I do not know the person and fear for the safety of myself and my clothes. If I were to trust any person to my laundry, I would need to know that they are a law abiding citizen and knowledgeable in cleaning clothes. Since I am living on a college campus, I would prefer that the person is also a student and lives in the same dorm building as I do. This will eliminate outsiders from entering a student residential hall and keep the worker accountable with the Office of Student Affairs. In addition, I would want the same gender to be doing my laundry.

Furthermore, since there will be some communication between me and the customer, I would want all my contact information to be hidden. This includes my last name and personal phone number, similar to what Uber does. If there must be some communication between me and the worker, it should be done in the app itself.

If there are any actions that are against the policies of the app, I should have an easy and fast way to file a complaint and/or contact a customer service member. This includes issues with how my clothing was handled or if the requirements of my laundry was completed correctly and any other issues with the app. I would also like to have a review system where I can post a review and see past reviews of a worker. This will make me feel much more comfortable when handing my clothes to someone I don’t know.

Glossary of Terms

Dasher: A user who chooses to provide their service for customers - this user would perform the laundry.

Customer: A user of Drip ‘n Dash who sets up a job for the Dasher - this user requests to have their laundry done.

Customer Rating: A score a dasher can set for the customer, within the range of zero to five stars. This rating is based on factors such as: whether or not the customer communicated having provided garments that require extra care, whether or not the customer was on time with respect to the arranged pickup and drop off time of the laundry, overall attitude towards the Dasher, etc.

Dasher Rating: A score within the range of zero to five stars of which a customer can assign to a dasher after having completed a transaction with that Dasher. This rating should be based on factors such as: the quality of the service provided by the dasher, whether or not the dasher was on time with respect to the arranged pickup and drop off time of the laundry, the customer service skills of the Dasher, etc.

Drip 'n Dash: a mobile application that provides a means for an on-demand, peer-to-peer laundry service among Rutgers students.

Job Request: A request that a Customer submits asking for a Dasher to complete their laundry.

Job Request Status: The status of a job request set by the Dasher. It specifies the stage of a customer's laundry in its cleaning. The stages are

Firestore: An online database, hosted by google, that we are using to store information on user' and jobs.

Preferred Dasher: For Customers who like a Dasher and would prefer them, it allows them to choose that specific Dasher.

System Requirements

Enumerated Functional Requirements

REQ #	PRIORITY	DESCRIPTION
0	5	As a Rutgers student that lives in a university dorm with public washers and dryers, I can register for Drip 'n Dash to be a customer or a dasher using a university email address
1	5	As a customer, I can submit a request for a dasher to do my laundry
2	5	As a customer, I can specify the time that the dasher will pickup and drop off my laundry
3	3	As a customer, when creating a job request I can view currently

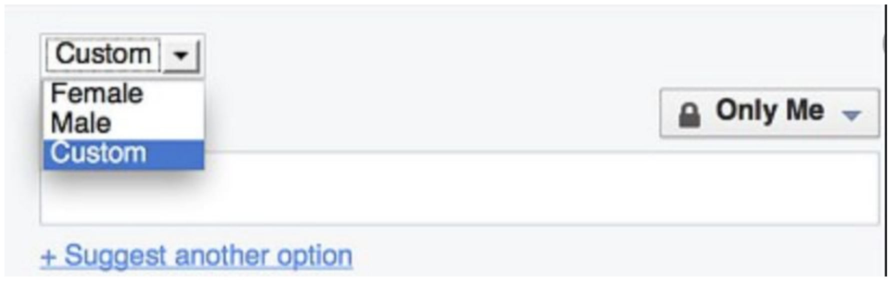
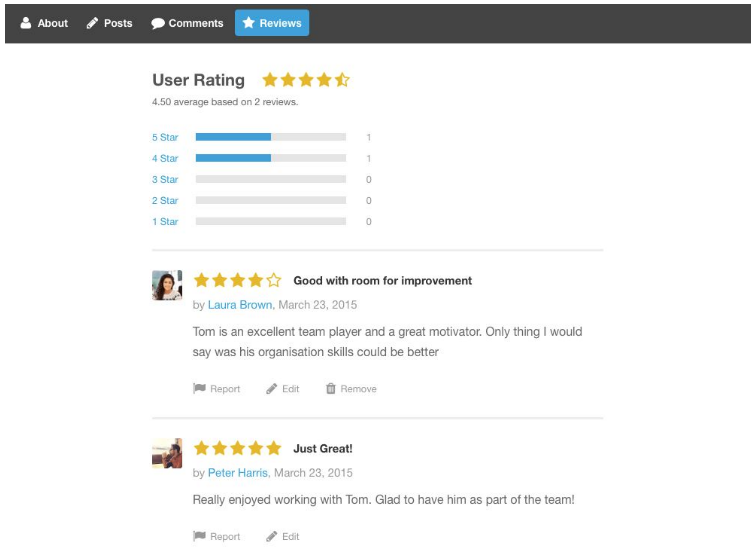
		online dashers and thus have a choice in dasher preference by choosing a dasher among them
4	5	As a dasher, I can be auto assigned a job upon request
5	4	As a dasher, I can accept or deny a customer's pending request for laundry depending on the presented customer information (name and star rating) and if I am able to meet the specified pickup and dropoff times
6	4	As a customer, I can see a presented dasher's information (star rating and name) once he/she accepts my request, and then choose to cancel or proceed with the request based on whether the dasher meets my criteria (e.g. a women customer might only want a woman dasher to do their laundry, and thus can reject any male dashers presented)
7	4	As a customer, I can leave a star rating and a detailed review for a dasher after completing a transaction with him/her
8	4	As a customer, I can cancel my request for laundry if it has not already been picked up
9	4	As a customer I can do a cashless transaction using a cashless service with google pay (Android) or apple pay (iOS).
10	3	As a customer, I can see basic information about the status of my in progress request for laundry. This includes information about when the dasher is outside to pick up my laundry, when my laundry is in the wash, in the dryer, etc.
11	3	As a dasher, I can notify the customer with basic status updates on their request for laundry
12	4	As a dasher, I can simultaneously complete multiple customers' requests for laundry and view a list of my in progress jobs
13	3	As a customer, I can review details regarding each of my completed laundry job requests in my completed request history
14	4	As a customer, I can dispute a transaction if the dasher did not successfully complete my request for laundry (stolen/damaged items, etc.)
15	4	As a customer, I can clear my "preferred dasher" selection if I decide to have my job request auto assigned to a random dasher after having already selected an online dasher

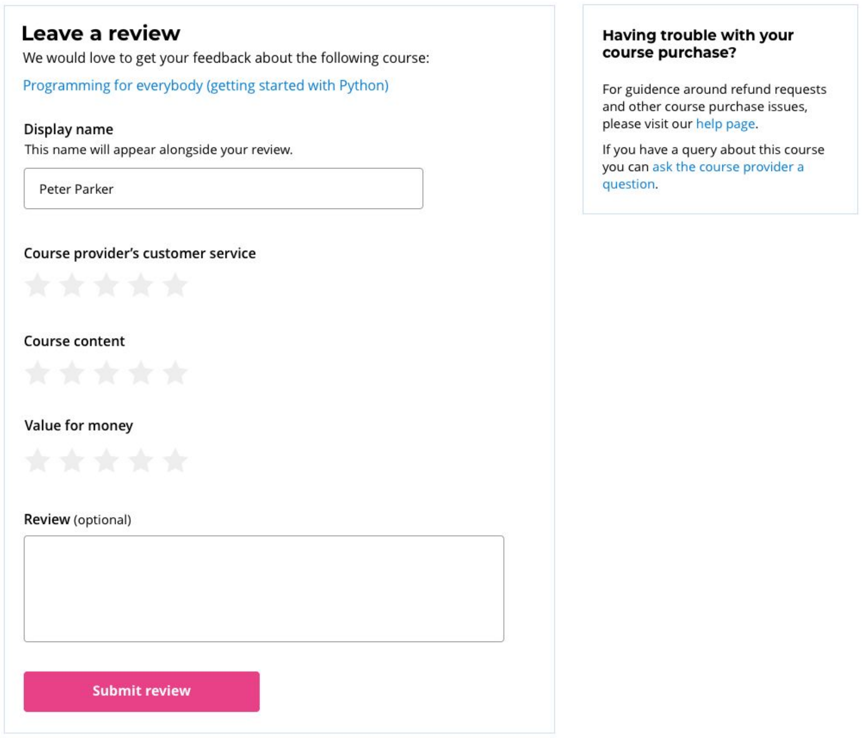
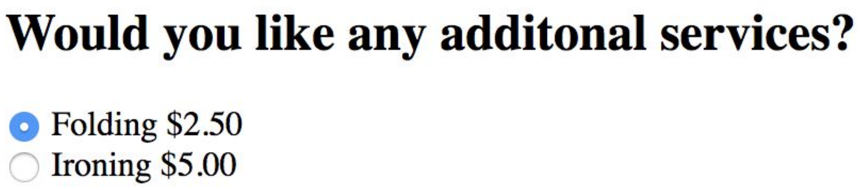
16	3	As a dasher, I can set my status to “accepting requests” or “not accepting requests” mode in order to be visible or invisible as an online dasher for customers requesting a job from a specific dasher
17	3	As a customer, I can enter comments specifying my job request (will be supplying my own detergent, load contains delicate fabric, etc.)

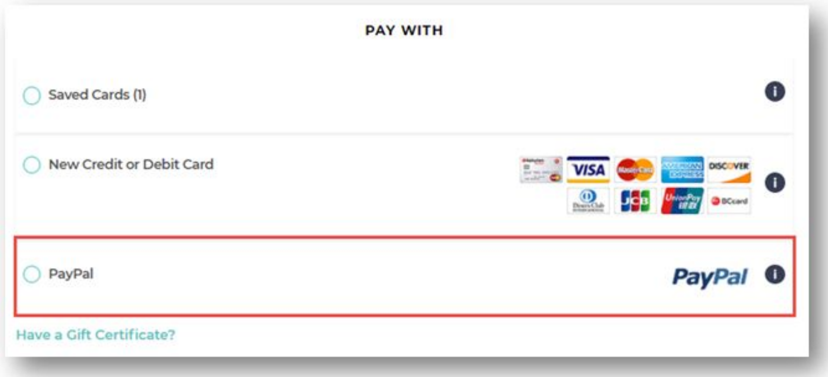
Enumerated Nonfunctional Requirements

REQ #	PRIORITY	DESCRIPTION
18	4	As a system, improve user usability to be as simple as possible for both dashers and users
19	5	As a system, maintain the privacy/confidentiality of certain information for both the customer and dasher
20	4	As a system, app optimization is needed to increase efficiency of the system and decrease latency overall.
21	5	Maintain accuracy with dorm/machine locations
22	3	Improve the maximum capabilities of the application while meeting the system requirements
23	3	Update/Maintain reviews for dashers to improve usability
24	2	System should be load balanced
25	5	As a system, the application must be accessible to most of today’s Android OS kernels
26	3	As a system, backups of data should be taken regularly
27	3	The system should have an easy and usable UX for users
28	5	The system should handle numbers of users without consuming too much resources
29	4	The system has to notify the policy of the app to the user
30	4	The system has to be easily maintainable so developers will be able to fix bugs

User Interface Requirements

REQ #	PRIORITY	DESCRIPTION
0	5	System needs to be able to access a list of available dashers in the users vicinity
1	5	The system should have a page with a filter where user can specify which gender they would prefer do their laundry  Custom Female Male Custom Only Me + Suggest another option
2	5	System requires a page that shows how many stars/approval rating designated dasher has  About Posts Comments Reviews Report Edit Remove User Rating ★★★★☆ 4.50 average based on 2 reviews. 5 Star 1 4 Star 1 3 Star 0 2 Star 0 1 Star 0 ★★★★☆ Good with room for improvement by Laura Brown, March 23, 2015 Tom is an excellent team player and a great motivator. Only thing I would say was his organisation skills could be better ★★★★★ Just Great! by Peter Harris, March 23, 2015 Really enjoyed working with Tom. Glad to have him as part of the team!
3	3	System also needs a page where the user can leave a review about how their laundry was done which would update the dasher's approval rating accordingly

		 **Leave a review** We would love to get your feedback about the following course: Programming for everybody (getting started with Python) **Display name** This name will appear alongside your review. Peter Parker **Course provider's customer service** ☆☆☆☆☆ **Course content** ☆☆☆☆☆ **Value for money** ☆☆☆☆☆ **Review** (optional) Submit review **Having trouble with your course purchase?** For guidence around refund requests and other course purchase issues, please visit our help page. If you have a query about this course you can ask the course provider a question.
4	5	The system should present a page, after the user selects a dasher, which shows any additional services (ironing, folding) users would like to have done for an extra cost.  **Would you like any additonal services?** [✓ Folding $2.50] Ironing $5.00
5	5	System must have a page where dasher can accept a customer's pending request for laundry if they can meet the specified pickup and drop off times
6	4	After all options and services are selected, the system must present a drop off/pick up location that is convenient for both user and dasher
7	5	Following the conclusion of a laundry service, the system should show a page that has payment options (venmo, credit card etc)

		 PAY WITH Saved Cards (1) New Credit or Debit Card PayPal Have a Gift Certificate?

Functional Requirements Specification

Stakeholders

Customers
Dashers
Dorm Resident
Rutgers
Any other colleges that would want to utilise this app
Any sponsors that are willing to run ads

Actors and Goals

Customer- Initiating: To be able to log in, and request the services provided by the application as they desire.

Dasher- Initiating: To be able to log in and take jobs that have been requested. Take jobs that fit the request criteria.

Job System- Participating: Allow customers to post requests for laundry, and to allow dashers to accept/be assigned to those requests.

Progress System- Participating: To provide a form of communication between Actors in order to present the status of the requested job.

Rating System- Participating: To present Customers with the quality of the Dasher. Alternatively, present the Dasher with how the Customer is.

Database- Participating: To store Actors data such as name, email address, password, campus, dorm, room number, and various other factors.

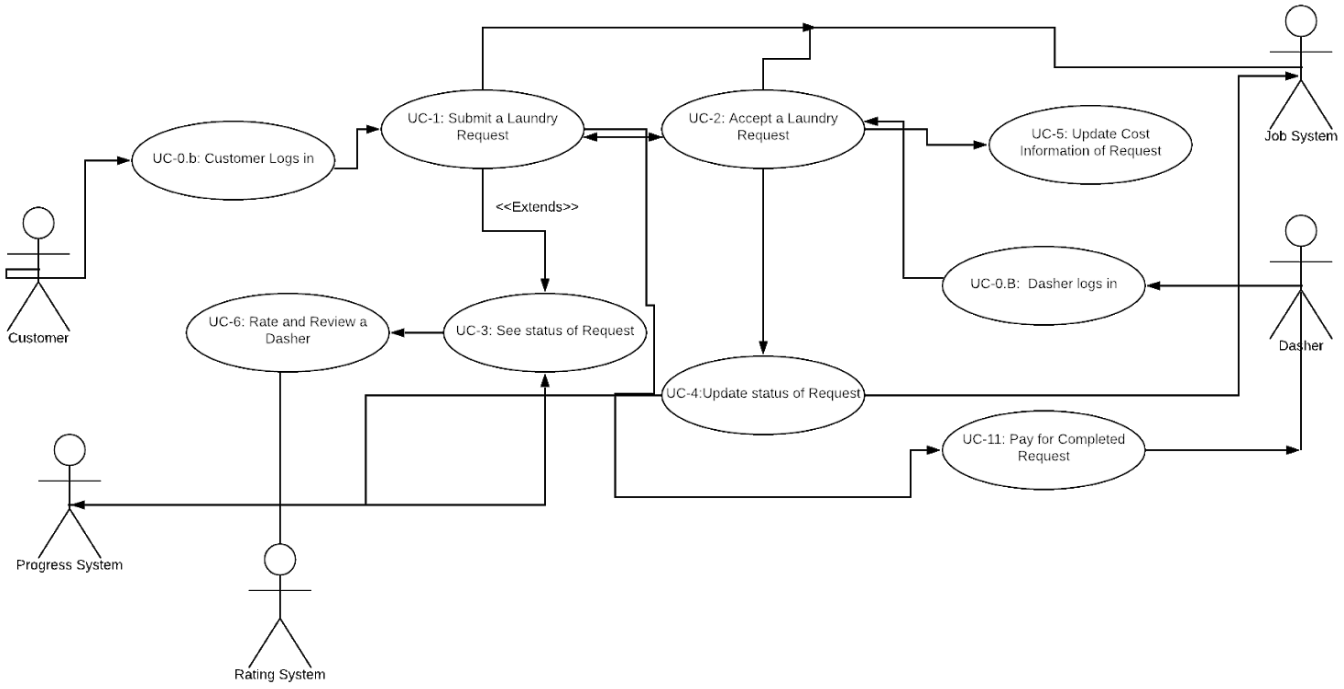
Use Cases

Casual Description

Use Cases	**Actors (bold->initiating, regular->participating)**	**Description (Actor's Goal)**
UC-0.A Register	● **Customer/Dasher** ● Database	User creation and login to be either a client or dasher
UC-0.B Login	● **Customer/Dasher** ● Database	User can login as a dasher or customer depending on what they registered as
UC-1 Submit a Laundry Request	● **Customer** ● Job System ● Database	Customers can submit a laundry request available to all dashers in their dorm. They can also specify specific instructions for the request
UC-2 Accept a Laundry Request	● **Dasher** ● Job System ● Database	Dashers can check if there are any pending requests for laundry in their dorm, and be assigned the oldest one
UC-3 See Status of Request	● **Customer** ● Progress System ● Job System ● Database	Customers can see a page with the detailed status of their request for laundry. This information will include whether its in the washer/dryer, and see information about the dasher assigned to the request
UC-4 Update Status of Request	● **Dasher** ● Job System ● Progress System ● Database	Dashers can update the progress they have made for each job they are working on.

UC-5 Update Cost Information of Request	● **Dasher** ● Job System ● Progress System ● Database	Once dashers reach the stage where the laundry is in the dryer, the dasher can update the actual cost required to complete the laundry request
UC-6 Rate and Review a Dasher	● **Customer** ● Rating System ● Progress System ● Database	Once a laundry request has been completed, a customer can write a written review and leave a star rating for the Dasher based on the quality of service
UC-7 Cancel a Job Request	● **Customer** ● Progress System ● Database	After submitting a laundry request, a customer can choose to cancel the request if their laundry has not already been picked up
UC-8 See Information about Past Jobs/Requests	● **Customer/Dasher** ● Job System ● Database	Upon dropping off the clients laundry the dasher can then rate the client which will appear in the client's module
UC-9 Request a Specific Dasher	● **Customer** ● Dasher ● Job System ● Database	Before submitting a laundry request, customers can see a list of all dashers in their dorm in "accepting requests" mode and send their request to a specific dasher in the list
UC-10 Change Request Acceptance Mode	● **Dasher** ● Database	Dashers can choose to allow specific requests for laundry to be sent to them, or not allow this. Note that dashers can still request jobs when not in "accepting requests" mode"
UC-11 Pay for Completed Request	● **Customer** ● Dasher ● Database	After the cost information has been updated by the dasher and the laundry has been dropped off, the customer can (must) pay for the laundry request

Use Case Diagram


UC-0.b: Customer Logs in
UC-1: Submit a Laundry Request
UC-2: Accept a Laundry Request
UC-5: Update Cost Information of Request
Job System
<<Extends>>
Customer
UC-0.B: Dasher logs in
Dasher
UC-6: Rate and Review a Dasher
UC-3: See status of Request
UC-4:Update status of Request
UC-11: Pay for Completed Request
Progress System
Rating System

Traceability Matrix

	REQ0	REQ1	REQ2	REQ3	REQ4	REQ5	REQ6	REQ7	REQ8
PW	5	5	5	3	5	4	4	4	4
UC0	X								
UC-1		X	X						
UC-2					X	X			
UC-3									
UC-4					X				
UC-5									
UC-6								X	
UC-7									X
UC-8				X			X		
UC-9									
UC-10									

UC-11									

	REQ9	REQ10	REQ11	REQ12	REQ13	REQ14	REQ15	REQ16	REQ17	Total
PW	4	3	3	4	3	4	4	3	3	
UC0										5
UC-1									X	10
UC-2				X						13
UC-3		X								3
UC-4										5
UC-5			X							3
UC-6					X					7
UC-7										4
UC-8						X				11
UC-9							X			4
UC-10								X		3
UC-11	X									4

Fully-Dressed Description

UC 0.A: CustomerRegister

- Actor: user
- Preconditions: the user is not registered for our application
- Postconditions: the user registers for our application and is prompted to the **CustomerHomePage**
- Flow of Events for Main Success

- 1 . User clicks the register button on the **SignInPage**
    - 2 . The user is prompted to the **RegisterInformationPage** where he/she can see information about customers/dashers
    - 3 . The user presses the register as a customer button and is prompted to the **CustomerRegisterPage** where he/she can input the information required and press the confirm registration button.
    - 4 . If the information entered is valid, he/she will be prompted to the **CustomerHomePage**

UC 0.A: Dasher Register

- Actor: user
- Preconditions: user is not registered for our application, or user has only registered a regular customer
- Postconditions: user registers for our application and is prompted to the **CustomerHomePage**
- Flow of events for main success:
    - 1 . User clicks the register button on the **SignInPage**
    - 2 . The user is prompted to the **RegisterInformationPage** where he/she can see information about customers/dashers
    - 3 . The user presses the register as customer button and is prompted to the **DasherRegisterPage**
    - 4 . If the information entered is valid, he/she will be prompted to the **DasherHomePage**

UC 0.A: CustomerLogin

- Actor: user
- Preconditions: the user is registered as a customer
- Postconditions: the user is prompted to the **CustomerHomePage**
- Flow of Events for Main Success
    - 1. User enters his/her information into the text fields on **SignInPage**
    - 2. User presses the log in button
    - 3. If the information entered is correct, the user will be prompted to the **CustomerHomePage**
    - 4. If the information entered is wrong, the user will be told to try again on **SignInPage**

UC 1: Submit a Laundry Request

- Actor: Customer
- Preconditions: the customer wants to post a request for laundry pick up
- Postconditions: customer's request is added to the database for dashers to accept
- Flow of events for main success:
    - 1 . User enters information about the request and presses the submit request button on the **CustomerHomePage**
    - 2 . Customer's request for laundry is added to the database and the in progress jobs section on the **CustomerHomePage** is updated to contain the request the customer submitted
- Flow of Events for Extensions
    - 1. User enters incorrect information about the request (

UC 2: Accept a Laundry Request

- Actor: Dasher
- Preconditions: dasher is logged in to application and on **DasherHomePage**
- Postconditions: customer's request is added to queue of in progress jobs, and dasher sees options to update progress bar of jobs
- Flow of events for main success:
    - 1. Dasher receives request and clicks on accept button
    - 2. Request is added to Dasher's in progress Jobs which is on **DasherHomePage**

UC 4: Update Status of Request

- Actor: Dasher, JobSystem, Progress System, Database
- Preconditions: The dasher wants to update the status of a job request.
- Postconditions: The status of the Job is updated in Firestore and a notification is sent to the customer indicating this progress update.
- Flow of Events for Main Success:
    - Dasher presses the button that sends the signal to the Firestore to update the job progress.
    - Firestore document changes the jobsInProgress collection with the document id UID.

UC-5: Update Cost Information of Request

- Actor: Dash, Job System, Progress System, Database
- Preconditions: Dasher has finished drying the laundry, and enters the amount of actual loads it took to dry all of the laundry.

- Postconditions: Actual cost of the job is updated in Firestore and is updated on both the Customer and Dasher end of the application.
- Flow of Events for Main Success
    - Dasher confirms the actual number of loads required to complete this job.
    - NUM_LOADS_ACTUAL field in Firestore collection jobsInProgress with document id UID is updated, and the total cost of the job is adjusted appropriately.
- Flow of Events for Extension
    - Dasher changes the value of the actual number of loads to a high number as a means to potentially gain more income.

UC-6: Rate and Review a Dasher

- Actor: Customer, Rating System, Progress System, Database
- Preconditions, Job is completed with the Laundry dropped off and the progress is updated in Firestore.
- Postconditions: Customers would rate the Dasher from zero to five stars based on their service and also input comments to serve as feedback for the dasher.
- Flow of Events for Main Success
    - Customer selects a star rating with the option of entering comments.
    - Flow of Events for Extension
        - Customer does not pick a star rating.

UC-7: Cancel a Job Request

- Actors: Customer, Progress System, Database
- Preconditions: Job request has been submitted and laundry has not been picked up yet.
- Postconditions: Job request has been removed from jobsInProgress collection in Firestore and Dashers cannot be given the option to accept this job request.
- Flow of Events for Main Success
    - Customer presses Cancel button and confirms the cancellation.
    - Firestore removes UID from the jobsInProgress collection and informs the Customer that their request has been cancelled.
- Flow of Events for Extension
    - Laundry has been picked up and a request cancellation has been requested.

UC - 8: See information about Past Job Requests

- Actors: Customer/Dasher, Job System, Database
- Preconditions: Job requests have been completed beforehand.

- Postconditions: Customer/Dasher would be able to view information about previous job requests including rating of the Dasher/Customer.
- Flow of Events for Main Success
    - Customer/Dasher enters profile/home page.
    - Customer/Dasher switches to a page containing Job request History.
- Flow of Events for Extension
    - Customer/Dasher has not processed any Job Requests so see any History.

UC - 9: Request Specific Dasher

- Actors: Customer, Dasher, Job System, Database
- Preconditions: Customer has requested a job at least once from a Dasher
- Postconditions: Customer will be able to request a specific Dasher to do their job
- Flow of Events for Main Success:
    - User logs in as Customer
    - Press button to request for a specific Dasher
- Flow of Events for Extension:
    - Dasher that Customer requested is not online
    - Customer does not press button for a specific Dasher and defaults to any Dasher

UC - 10: Change Request Acceptance Mode

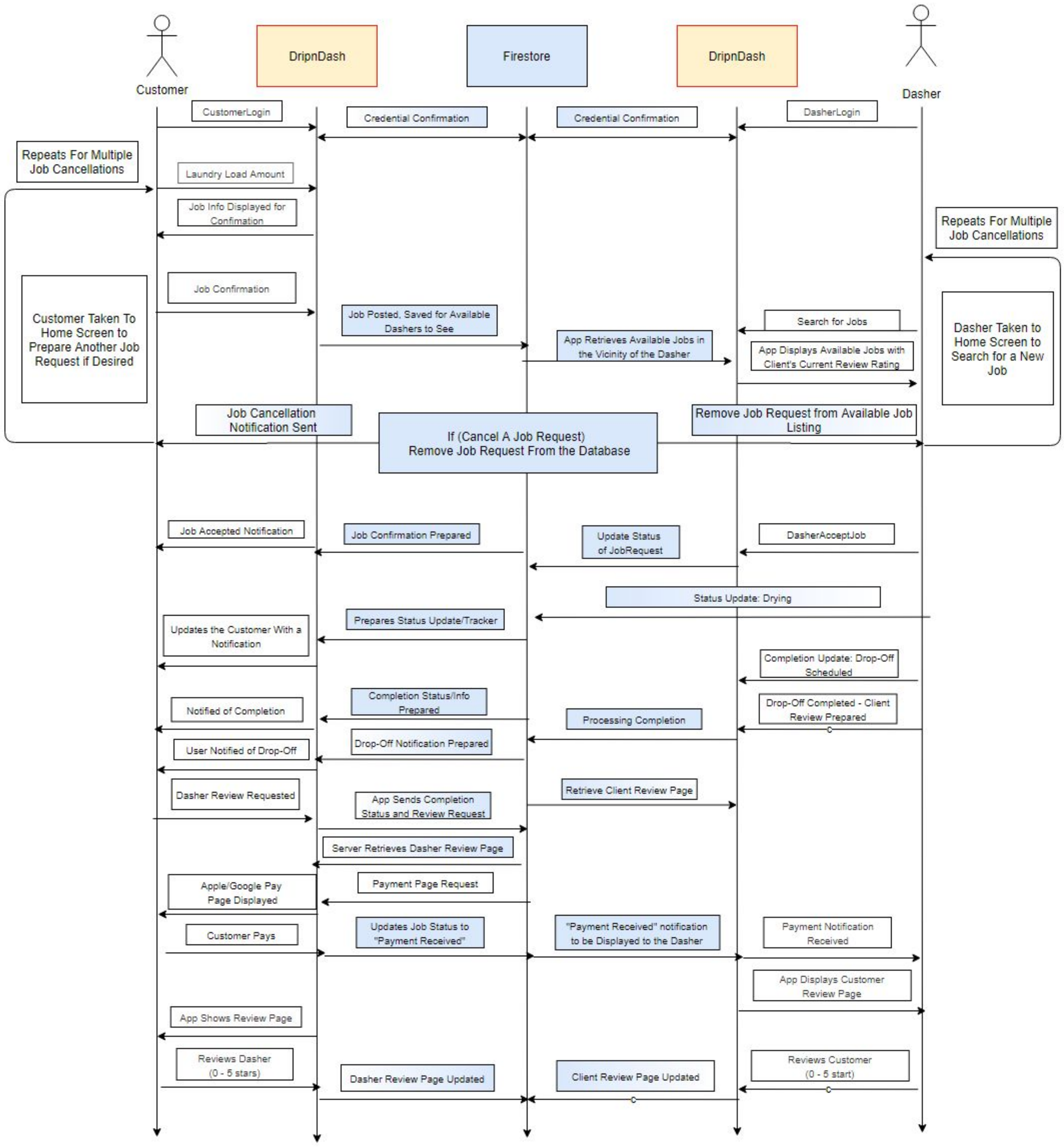
- Actors: Dasher, Database
- Preconditions: User has registered as a Dasher
- Postconditions: Dasher presses a button to be able to accept specific job requests from customers, but will be able to accept general jobs as well.
- Flow of Events for Main Success
    - User logs in as a Dasher
    - Button pressed to enable Specific Job Accepting.
- Flow of Events for Extension
    - Dasher does not press the button to enable Specific Job Accepting.

UC - 11: Pay for Completed Request

- Actors: Customer
- Preconditions: User has registered as a customer
- Postconditions: Customer confirms the job receipt and has their bank account information already set up with Apple Pay or Google Pay.
- Flow of Events for Main Success
    - Bank information already set up with respective cashless service and is active.

    - Dasher confirms receipt and predicted cost.
- Flow of Events for Extension
    - Customer’s bank account has an error completing the transaction
    - Customer disputes receipt cost.

System Sequence Diagram



Customer
DripnDash
Firestore
DripnDash
Dasher
CustomerLogin
Credential Confirmation
Credential Confirmation
DasherLogin
Repeats For Multiple Job Cancellations
Laundry Load Amount
Job Info Displayed for Confimation
Repeats For Multiple Job Cancellations
Job Confirmation
Customer Taken To Home Screen to Prepare Another Job Request if Desired
Job Posted, Saved for Available Dashers to See
App Retrieves Available Jobs in the Vicinity of the Dasher
Search for Jobs
App Displays Available Jobs with Client's Current Review Rating
Dasher Taken to Home Screen to Search for a New Job
Job Cancellation Notification Sent
If (Cancel A Job Request) Remove Job Request From the Database
Remove Job Request from Available Job Listing
Job Accepted Notification
Job Confirmation Prepared
Update Status of JobRequest
DasherAcceptJob
Status Update: Drying
Prepares Status Update/Tracker
Updates the Customer With a Notification
Completion Update: Drop-Off Scheduled
Notified of Completion
Completion Status/Info Prepared
Processing Completion
Drop-Off Completed - Client Review Prepared
User Notified of Drop-Off
Drop-Off Notification Prepared
Dasher Review Requested
App Sends Completion Status and Review Request
Retrieve Client Review Page
Server Retrieves Dasher Review Page
Apple/Google Pay Page Displayed
Payment Page Request
Customer Pays
Updates Job Status to "Payment Received"
"Payment Received" notification to be Displayed to the Dasher
Payment Notification Received
App Displays Customer Review Page
App Shows Review Page
Reviews Dasher (0 - 5 stars)
Dasher Review Page Updated
Client Review Page Updated
Reviews Customer (0 - 5 start)

Effort Estimation using Use Case Points

- UC 0-A Register:
    - 1 click: select register account option on application sign in page
    - 1 click: select either customer or dasher on “Choose a Type of User” page
    - 7 clicks: enter in every data text field on registration page
    - 1 click: click on submit button to finalize account registration information
    - **Total 10 clicks: 7/10 clerical data entry, 3/10 UI navigation**
- UC 0-B Login:
    - 2 clicks: enter in username and password into text fields on sign in page
    - 1 click: click on login button to submit account information
    - **Total 3 clicks: 2/3 clerical data entry, 1/3 UI navigation**
- UC-1 Submit a Laundry Request:
    - 1 click: press home button if not already there (submit request page)
    - 2 clicks: enter in the text fields: number of loads and instructions
    - 1 click: press submit request button
    - 1 click: click accept or decline on presented dasher match
    - **Total 5 clicks: 2/5 clerical data entry, 3/5 UI navigation**
- UC-2 Accept a Laundry Request:
    - 1 click: press job accepting mode button
    - 1 click: press assign me a new job button
    - 1 click: click accept or decline on presented customer and job description
    - **Total 3 clicks: 1/3 clerical data entry, 2/3 UI navigation**
- UC-3 See Status of Request:
    - 1 click: click on job you want to see status of under “in progress requests” table on the home page
    - **Total 1 click: 1/1 UI navigation**
- UC-4 Update Status of Request:
    - 1 click: click on job you want to update status of in “in progress jobs” table on the home page
    - 1 click: set status of job to “on way for pickup”
    - 1 click: set status of job to “picked up laundry”
    - 1 click: set status of job to “laundry in washer”
    - 1 click: set status of job to “finished washing”
    - 1 click: set status of job to “laundry in dryer”
    - 1 click: set status of job to “finished drying”
    - 1 click: set status of job to “on way for drop off”

- **Total 8 clicks: 7/8 clerical data entry, 1/8 UI navigation**

- UC-5 Update Cost Information of Request:
  - 1 click: enter in text field of total cost incurred
  - 1 click: enter in text field of actual number of loads done
  - 1 click: press submit
  - **Total 3 clicks: 2/3 clerical data entry, 1/3 UI navigation**
- UC-6 Rate and Review a Dasher:
  - 1 click: enter in star rating text field
  - 1 click: enter in comment section
  - 1 click: press submit
  - **Total 3 clicks: 2/3 clerical data entry, 1/3 UI navigation**
- UC-7 Cancel a Job Request:
  - 1 click: on laundry request status, click "cancel request"
  - 1 click: click on "yes cancel it"
  - **Total 2 clicks: 1/2 clerical data entry, 1/2 UI navigation**
- UC-8 See Information about Past Jobs/Requests:
  - 1 click: click on "past requests"/"past jobs" page at the bottom screen toolbar
  - **Total 1 click: 1/1 UI navigation**
- UC-9 Request a Specific Dasher:
  - 1 click: when submitting a request (UC-1), click on a dasher in the "Dashers Accepting Requests" table to have them set as your dasher preference
  - 1 click: click on the "clear dasher preference" if customer wishes to have their job auto assigned rather than to a selected online dasher
  - **Total 2 clicks: 1/2 clerical data entry, 1/2 UI navigation**
- UC-10 Change Request Acceptance Mode:
  - 1 click: on home page click "accepting requests"
  - 1 click: on home page click "not accepting requests"
  - **Total 2 clicks: 2/2 UI navigation**
- UC-11 Pay for Completed Request:
  - 1 click: click confirm on pop up receipt screen
  - **Total 1 click: 1/1 UI navigation**

Domain Analysis

Domain Model

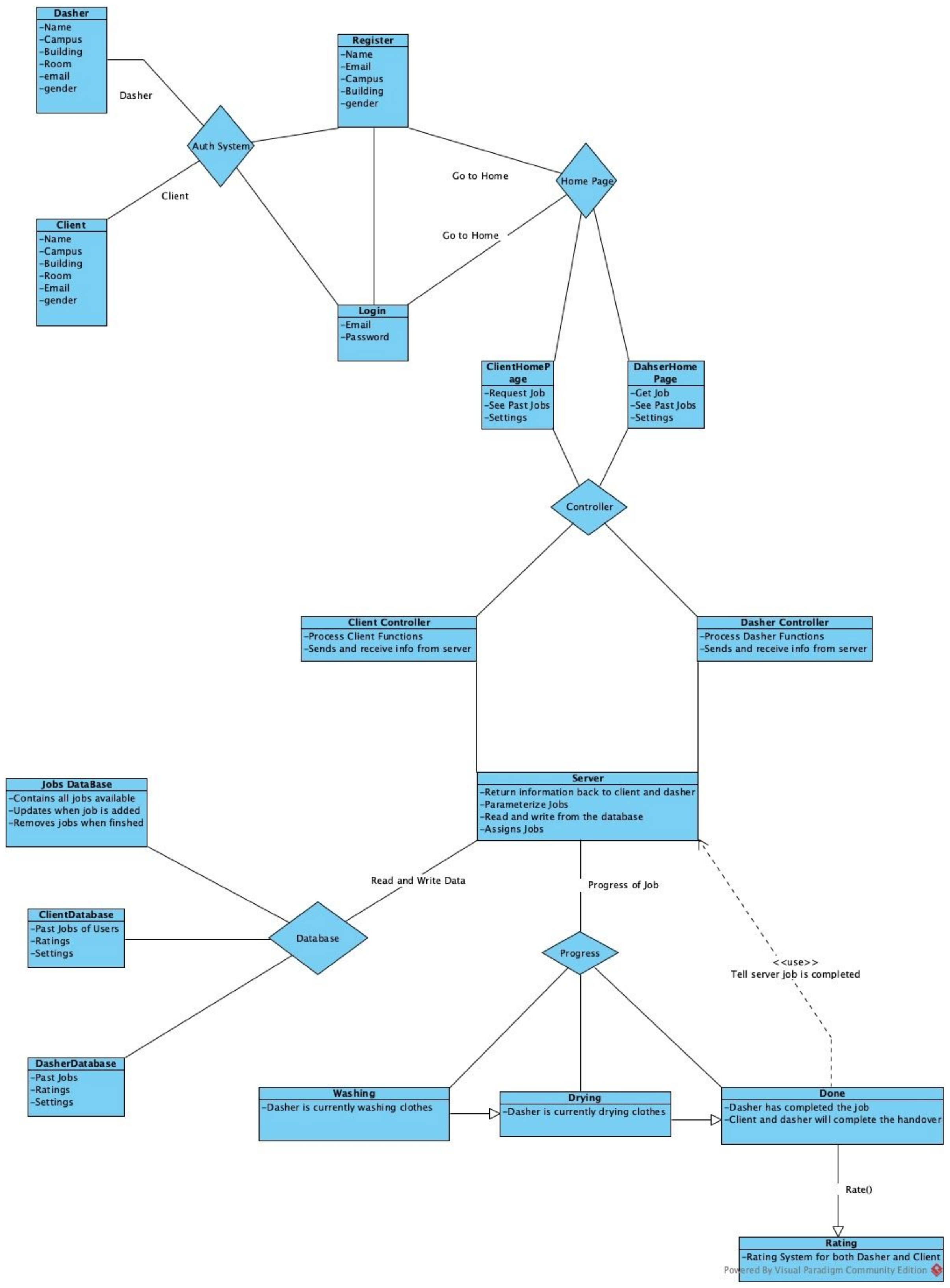
To find the domain model the group first thought of the key features that we wanted to implement. Once the features were established the next step is to figure out how we are going to implement them. Fortunately for us most of the methods and classes call on each other making the project less complex than others. Although implementing such features may seem tedious, the way that we went about it was further from it.

The first step is understanding the fields for both Client's and Dashers. This includes Name, Email, Password, Campus, Building, Room Number, and Gender. All the information will be saved through our Database in Firestore. These fields will be entered either through the registration page. The Authentication system handles the users who either login or decided to join with a registration page. Once the system determines that the users are valid they will be presented to a home page. The Home page is where all the functionality is going to be accessed through. When the user wants to use a functionality a controller will process the input and output to ensure that the desired outcome is valid.

Since the premise relies on real time and updated data the server will pull from a centralized database. This makes the database easier to implement and also less complex. The Database will include all job posting alongside user info. Once a job is requested by a Client the server will write the job offering on a Job listing portion of the database that Dashers can view. Once the job is accepted by a Dasher it is taken out of the listing and the Client will be notified.

The job then will pass through a progress function. The Dasher will update the system on the status of the job. This status can be viewed in the clients homepage.

Once the job is completed and the clothes are dropped/picked up a rating system will appear for both the Client and Dasher. The rating system is used to ensure that both the Clients and Dashers receive the best experience and continue to use the service. This cycle continues as long as there are students with dirty clothes.


Dasher
-Name
-Campus
-Building
-Room
-email
-gender
Dasher
Register
-Name
-Email
-Campus
-Building
-gender
Auth System
Client
Client
-Name
-Campus
-Building
-Room
-Email
-gender
Go to Home
Home Page
Go to Home
Login
-Email
-Password
ClientHomePage
-Request Job
-See Past Jobs
-Settings
DahserHomePage
-Get Job
-See Past Jobs
-Settings
Controller
Client Controller
-Process Client Functions
-Sends and receive info from server
Dasher Controller
-Process Dasher Functions
-Sends and receive info from server
Server
-Return information back to client and dasher
-Parameterize Jobs
-Read and write from the database
-Assigns Jobs
Jobs DataBase
-Contains all jobs available
-Updates when job is added
-Removes jobs when finshed
Read and Write Data
Progress of Job
ClientDatabase
-Past Jobs of Users
-Ratings
-Settings
Database
Progress
<<use>>
Tell server job is completed
DasherDatabase
-Past Jobs
-Ratings
-Settings
Washing
-Dasher is currently washing clothes
Drying
-Dasher is currently drying clothes
Done
-Dasher has completed the job
-Client and dasher will complete the handover
Rate()
Rating
-Rating System for both Dasher and Client
Powered By Visual Paradigm Community Edition

Concept Definitions:

Responsibility Description	**Type**	**Concept Name**
Store all authentication information about registered customers and dashers, in progress laundry requests, master list of all completed laundry requests, etc.	K	Database <<entity>>
Allow a registered user (customer/dasher) to sign in by entering his/her email and password, and provide link to register for the application	D	SignIn <<boundary>>
Provide information to an unregistered user what a customer is and what a dasher is in our application.	D	RegisterInformation <<boundary>>
Allow an unregistered user to enter his/her information and sign up for our application as a customer.	D	CustomerRegister <<boundary>>
Allow an unregistered user to enter his/her information and sign up for our application as a dasher	D	DasherRegister <<boundary>>
Determine if the information a user entered to register as a customer is valid, and initialize customer's data in the database (Firestore). Interface for the application and reads/writes to customer's information in the database	D	CustomerFirestore <<entity>>
Determine if the information a user entered to register as a dasher is valid, and initialize customer's data in the database (Firestore). Interface for the application and reads/writes to dasher's information in the database	D	DasherFirestore <<entity>>
Container for customer's data frequently used by other parts of the system (firstName, lastName, email, dorm, dormRoom, etc.)	K	Customer <<entity>>
Container for dasher's data frequently used by other parts of the system (firstName, lastName, email, dorm, dormRoom, etc.)	K	Dasher <<entity>>
Allow a customer to enter information about his/her request for laundry and submit it, and render an interactive list with all of a customer's in progress jobs	D	CustomerHome <<boundary>>
Present detailed information about the current status of a customer's in progress laundry request including (CURRENT_STAGE in ["pending dasher assignment", "dasher outside for pickup", etc.], estimated drop off time, link to DasherProfile of dasher assigned to the request,	D/K	CustomerJobStatus <<boundary>>

etc.)		
Present an interactive list of all a customer's completed requests for laundry with basic information about each request	D/K	CustomerJobsHistory <<boundary>>
Present detailed information about a customer's past laundry request including (timeStamp the request was submitted, timeStamp the request was completed, dasher that completed the request, amount paid, etc.)	D	CustomerPastJobInfo <<boundary>>
Allow a dasher to accept new jobs, and render a list with basic information about the current status of the dasher's in progress jobs.	D	DasherHome <<boundary>>
Allow a dasher to update the current status of a customer's laundry request (CURRENT_STATUS in ["Outside for Pickup", "Laundry in Washer", "Outside for Drop Off", etc.]	D	DasherJobStatus <<boundary>>
Present an interactive list of all a dasher's completed jobs with basic information about each job	D	DasherJobsHistory <<boundary>>
Present detailed information about a dasher's past laundry request including (timeStamp the request was submitted, timeStamp the request was completed, dasher that completed the request, amount paid, etc.)	D	DasherPastJobInfo <<boundary>>
Container for all information related to a customer's job/laundry request including (jobID, number of loads, customer's dorm, customer's dorm room, etc.)	K	JobRequest <<entity>>
Handle reads/writes about JobRequest information to the database (Firestore). Interface for the application and the JobRequest portion of the database.	D	JobRequestFirestore <<entity>>
Listen for changes to the status of JobRequests in the database (updates made by a dasher) and notify parts of the system that need this information	D	DatabaseListener <<entity>>

Concept Pair	Association Description	Association Name
SignIn <--> Database	SignIn sends user's authentication information and Database returns whether the user is registered as a customer, dasher, or neither	sendsAuthData
SignIn -> RegisterInformation	If a user clicks the register button, SignIn presents the RegisterInformation page	presentRegInfo
RegisterInformation -> (Customer/Dasher)Register	If a user clicks the (customer/dasher) register button, RegisterInformation presents the (Customer/Dasher)Register page	present(C/D)Reg
CustomerRegister -> CustomerFirestore	CustomerRegister sends a user's inputted registration information to CustomerFirestore, which returns isValidRegistration = true if the form is valid (email domain @rutgers.edu, password meets security requirements, etc.), else it returns isValidRegistration = false	validatesRegData
CustomerFirestore -> Database	CustomerFirestore sends the registration information it received from CustomerRegister to the Database, and creates an entry in the Database containing all the customer's data	initCustomerData
DasherRegister -> DasherFirestore	DasherRegister sends a user's inputted registration information to DasherFirestore, which returns isValidRegistration = true if the form is valid (email domain @rutgers.edu, password meets security requirements, etc.), else it returns isValidRegistration = false	validatesRegData
DasherFirestore -> Database	DasherFirestore sends the registration information it received from DasherRegister to the Database, and creates an entry in the Database containing all the dasher's data	initDasherData
CustomerHome -> JobRequestFirestore	CustomerHome sends the information a customer submitted about a job request, and JobRequestFirestore interfaces with the database to store the request	submitJobRequest
JobRequestFirestore -> Database	JobRequestFirestore writes the information about a customer's JobRequest in the database (@dorms/ Dasher.dormName/jobsPendingAssignment/jobID and @jobsInProgress/jobID)	writeJobRequest
DasherHome ->	When a user enters job accepting mode, DasherHome	getJobRequest

JobRequestFirestore	asks JobRequestFirestore to assign the dasher the oldest pending request to which he/she has access	
JobRequestFirestore -> Database	When prompted by DasherHome, JobRequestFirestore reads all the pending JobRequests in the database (@dorms/ Dasher.dormName/jobsPendingAssignment) to which the dasher has access	readJobRequests
DatabaseListener -> Database	DatabaseListener checks for changes/updates to status information in a JobRequest document in the database (@jobsInProgress/JobRequest.jobID)	checkForUpdates
DatabaseListener ->CustomerJobStatus	DatabaseListener notifies CustomerJobStatus about any changes to the current status of a customer's JobRequest	notifyWithUpdates
DasherJobStatus -> JobRequestFirestore	When a dasher makes an update to the status of a customer's JobRequest, DasherJobStatus sends the updated information to JobRequestFirestore	updateJobRequest
JobRequestFirestore -> Database	When prompted by DasherJobStatus, JobRequestFirestore writes the updates to a JobRequest in the database (@inProgressJobs/JobRequest.jobID)	writeUpdates
(C/D)Home -> (C/D)JobStatus	When a customer/dasher selects a job in the list of in progress jobs on the (Customer/Dasher)Home, (C/D)Home presents the (C/D)JobStatus for that specific job	presentJobStatus
(C/D)JobsHistory -> (C/D)JobInformation	When a customer/dasher selects a job in the list his/her completed jobs on the (Customer/Dasher)JobsHistory, (C/D)JobsHistory presents the (C/D)PastJobInformation page for the specific job he/she clicked	presentPastJobInfo

Concept	Attribute	Attribute Description
Database	storedData	All data stored in the database including (customers, dashers, inProgressJobs, etc.)
SignIn	authData	A user's authentication information, currently it represents the user's email and password
(Customer/Dasher)Register	- registrationData - isValidRegistration	- The required information a user submits to register as a customer/dasher (name, email, etc.) - Boolean value stating whether the the information the user submitted is valid
CustomerHome	- inProgressJobs - JobRequestInfo	- array of JobRequest.jobID's for all of a customer's current in progress jobs - Information about a job request a customer submits
DasherHome	- inProgressJobs - isAcceptingJobs	- array of JobRequest.jobID's for all a dasher's current in progress jobs - Boolean value stating whether dasher is currently accepting jobs ("in job accepting mode")
(Customer/Dasher)JobStatus	- jobID - JobRequest	- jobID of the JobRequest this JobStatus page is showing - JobRequest object with all current information about the status of the job, and other relevant details
(Customer/Dasher)JobsHistory	- completedJobs	- array of JobRequest objects used to populate the information on the interactive list of a customer/dasher's past jobs on this page
DatabaseListener	- jobRequestUpdates	- updates to a JobRequest read from the Database

Traceability Matrix

	UC-0	UC-1	UC-2	UC-3	UC-4	UC-5	UC-6	UC-7	UC-8	UC-9	UC-10	UC-11
Database	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**
SignIn	**X**											
RegisterInformation	**X**											
CustomerRegister	**X**											
Dasherregister	**X**											
CustomerFirestore	**X**											
DasherFirestore	**X**											
Customer	**X**											
Dasher	**X**											
CustomerHome		**X**		**X**								
CustomerJobStatus				**X**								
CustomerJobsHistory									**X**			
CustomerPastJobInfo									**X**			
DasherHome			**X**		**X**							
DasherJobStatus						**X**						
DasherJobsHistory									**X**			
DahserPastJobInfo									**X**			
JobRequest		**X**								**X**	**X**	
JobRequest Firestore				**X**	**X**	**X**						
DatabaseListener				**X**	**X**			**X**				

System Operation Contracts

Name: login()
Responsibility: Use the database to control account systems and decipher between Customer and Dashers

Cross-References: UC-1
Output: The system will present the user with an UI that will ask them to submit the corresponding credentials
Pre-Conditions: The account name and password must be valid in the database before allowing the account to gain access to the system
Post-Conditions:

- ➔ Customers should have access to request jobs done and see the status of their requested job
- ➔ Dashers should be able to

Name: JobCreation()
Responsibility: Creates jobs to be sent out to the Dashers.
Cross-References: UC-2, UC-4
Output: The system presents the Dasher with a job that a Customer requested. The Dasher could either keep the job, or decline it.
Pre-Conditions: Dasher account must be logged on in order to receive a new job
Post-Conditions:

- ➔ The Dasher will receive the job, and from there they can update the progress on their job

Name: JobUpdates()
Responsibility : Updates the Customer with the progress the Dasher has made with their laundry.
Cross-References: UC-5
Output: The system will update the progress bar on the Customer's end so that they can see how far along the Dasher is with their laundry.
Pre-Conditions:

- ➔ The Customer must be logged in and has to have requested a job that was accepted by a Dasher.
- ➔ Dasher must be logged in and has to have accepted a job request from a Customer.

Post-Conditions:

- ➔ On the Customer's end the progress bar will be updated to show what step the Dasher is currently on
- ➔ On the Dasher end they have to press a button that will update the the job to reflect what step they are on until the job is finished

Name: JobCompletion()
Responsibility: The system informs the Customer that their request has been completed and is ready to be dropped off
Cross-References: UC-6
Output: Once the system sees that the Dasher has gone through all of the steps in order to

complete the job, it will notify the Customer that their job is ready to be dropped off
Pre-Conditions: Customer and Dasher must be logged in. The Dasher has to have completed the previous steps in order to notify the Customer that the job is ready to be dropped off
Post-Conditions: The Customer will be presented with a notification that the Dasher is on their way to drop off the laundry they had requested to be cleaned.

Name: Rating()
Responsibility: A rating system for both the Customer and the Dashers to tell the quality of the user.
Cross-References: UC-7, UC-8
Output: The system will display the rating of the Dasher to the Customer. Additionally after a job is completed the Customer can leave a rating stating if the Dasher did a good job or not. Alternatively, the Dasher can also rate Customers.
Pre-Conditions: Customer or Dasher must be logged on.

- ➔ Customer must have requested at least 1 job in order to receive a rating
- ➔ Dashers must have completed at least 1 job in order to receive a rating

Post-Conditions: .

- ➔ Customers will receive a rating based on how they are as a customer.
- ➔ Dashers will receive a rating which would signify their quality as a Dasher.

Name: Payment()
Responsibility: A payment system for so that the Customer can pay the Dasher for using their services.
Cross-References: UC-3
Output: The system will be able to take payment from the Customer in order to pay the Dasher.
Pre-Conditions: Customer or Dasher must be logged on.

- ➔ Customers must utilize Apple Pay or Google Wallet services.
- ➔ Dashers must utilize Apple Pay or Google Wallet services.

Post-Conditions: .

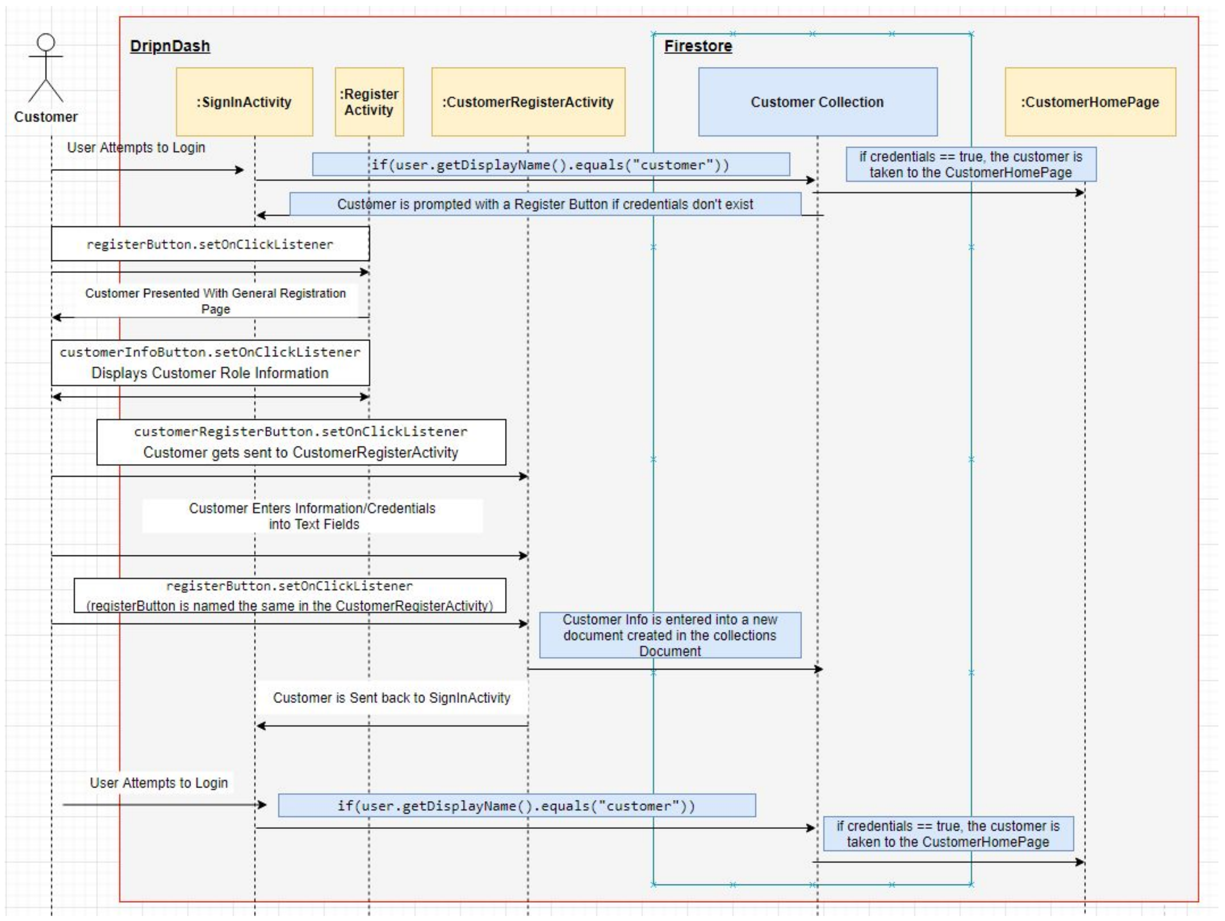
- ➔ Customers will have the money taken from their preferred payment option.
- ➔ Dashers will receive payment to their preferred payment option.

Mathematical Model

The main model we are using for this project is based on pricing, represented by a function based on the number of loads as well as the type of machine used during the washing and drying process. The end result should be greater than the minimum wage, since the whole process should take at least an hour to complete. We would then separate this fee into a flat fee for machines, and the rest to (whatever we decide to price, typically between \$5 and \$10). Our function for price is represented as: $Price(n) = (n * machine\ costs) + flat\,fee + convenience\,fees$, where n is the number of loads required to complete the request.

Interaction Diagram

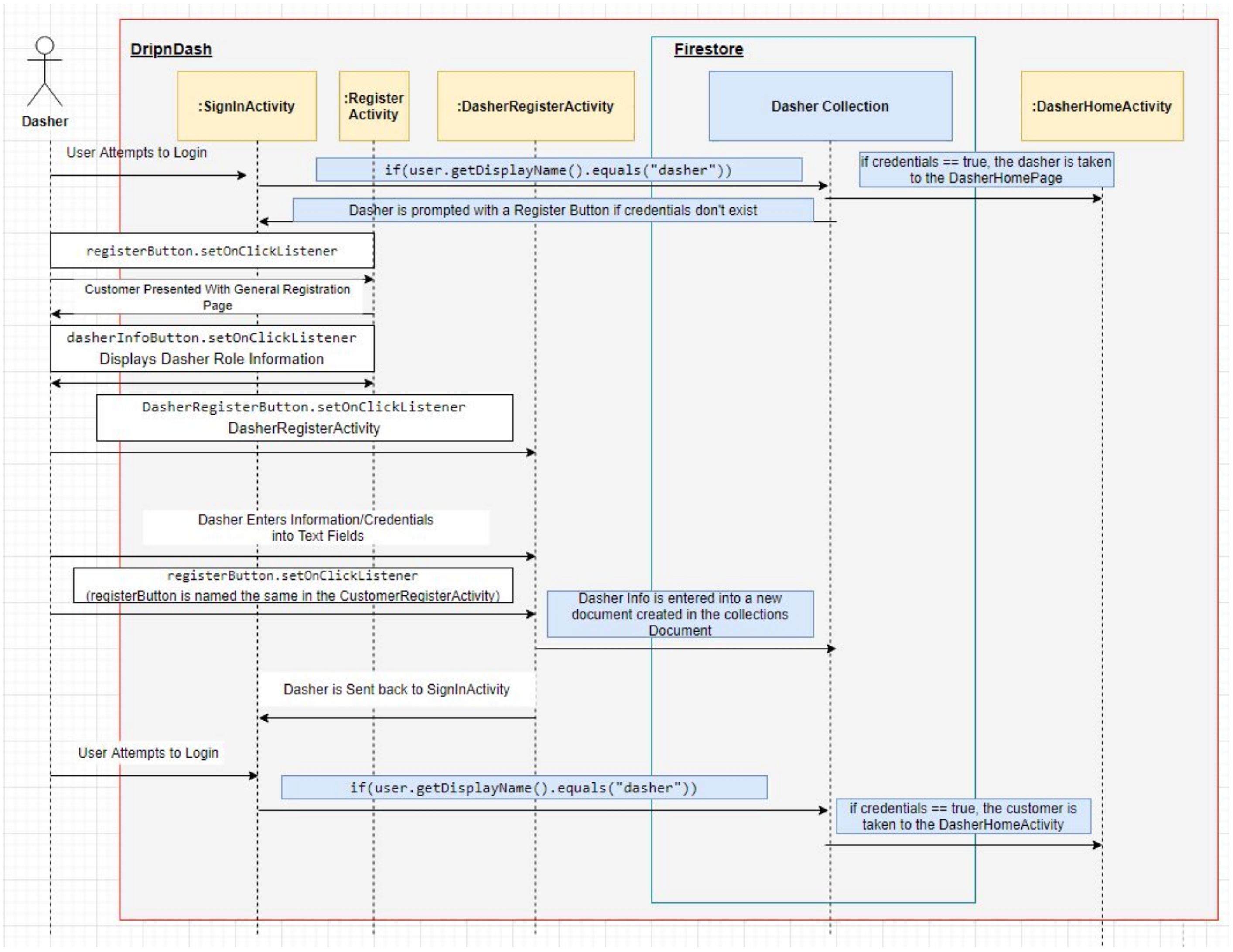
UC0.A: Customer Registration


Customer
DripnDash
Firestore
:SignInActivity
:Register Activity
:CustomerRegisterActivity
Customer Collection
:CustomerHomePage
User Attempts to Login
if(user.getDisplayName().equals("customer"))
if credentials == true, the customer is taken to the CustomerHomePage
Customer is prompted with a Register Button if credentials don't exist
registerButton.setOnClickListener
Customer Presented With General Registration Page
customerInfoButton.setOnClickListener
Displays Customer Role Information
customerRegisterButton.setOnClickListener
Customer gets sent to CustomerRegisterActivity
Customer Enters Information/Credentials into Text Fields
registerButton.setOnClickListener
(registerButton is named the same in the CustomerRegisterActivity)
Customer Info is entered into a new document created in the collections Document
Customer is Sent back to SignInActivity
User Attempts to Login
if(user.getDisplayName().equals("customer"))
if credentials == true, the customer is taken to the CustomerHomePage

Design Principles:

This figure shows the design sequence diagram for Use Case:Customer Registration. Once the user enters the application, the user will need to register and will have options to choose between “Register as a Customer” or “Register as a Dasher " and put the user's information in order to be registered as Customer. When the customer completes the registration process, a document in Firestore is created under the Customer Collection and the application will redirect the customer back to the Sign In page and the customer will be able to sign in to the Customer Home Page. Interface Segregation Principle is used because Drip’n dasher will have two different types of users, customers and dashers. Therefore we provide two different client specific interfaces to make the application easier to use.

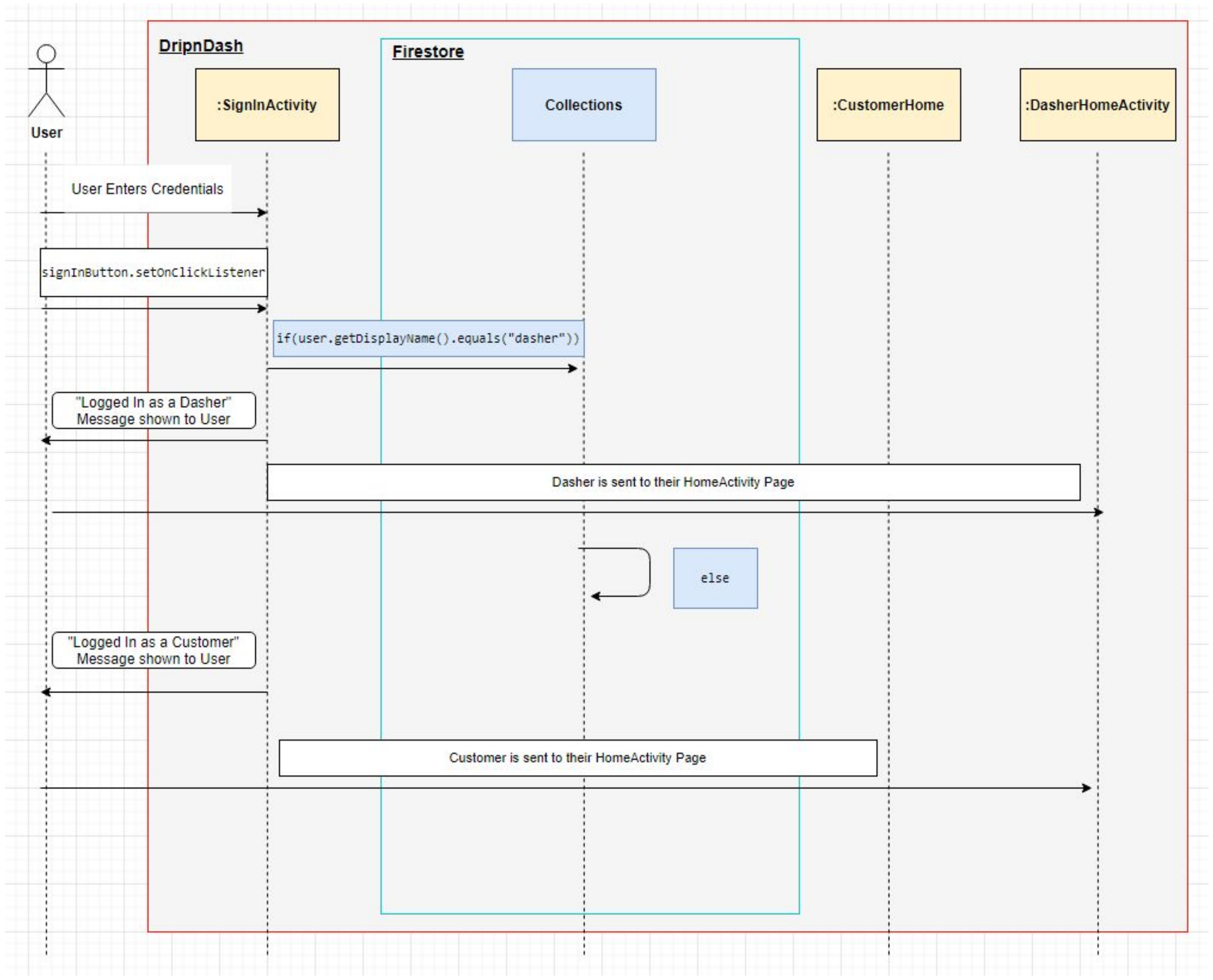
UC0:A: Dasher Registration


DripnDash
Firestore
Dasher
:SignInActivity
:Register Activity
:DasherRegisterActivity
Dasher Collection
:DasherHomeActivity
User Attempts to Login
if(user.getDisplayName().equals("dasher"))
if credentials == true, the dasher is taken to the DasherHomePage
Dasher is prompted with a Register Button if credentials don't exist
registerButton.setOnClickListener
Customer Presented With General Registration Page
dasherInfoButton.setOnClickListener
Displays Dasher Role Information
DasherRegisterButton.setOnClickListener
DasherRegisterActivity
Dasher Enters Information/Credentials into Text Fields
registerButton.setOnClickListener
(registerButton is named the same in the CustomerRegisterActivity)
Dasher Info is entered into a new document created in the collections Document
Dasher is Sent back to SignInActivity
User Attempts to Login
if(user.getDisplayName().equals("dasher"))
if credentials == true, the customer is taken to the DasherHomeActivity

Design Principles:

This figure shows the design sequence diagram for Use Case:Dasher Registration. Once the user enters the application, the user will need to register and will have options to choose between “Register as a Customer” or “Register as a Dasher ". In this scenario, the “Register as a Dasher” button will be pressed. If the dasher completes the registration process, a document in the Dasher collection will be created in Firestore and the system will redirect the user to the Sign In page and the dasher will be able to sign in to the Home Page. For this diagram, Interface Segregation Principle is also used because this diagram is simply identical to the above use case’s diagram, although the role of the user is switched..

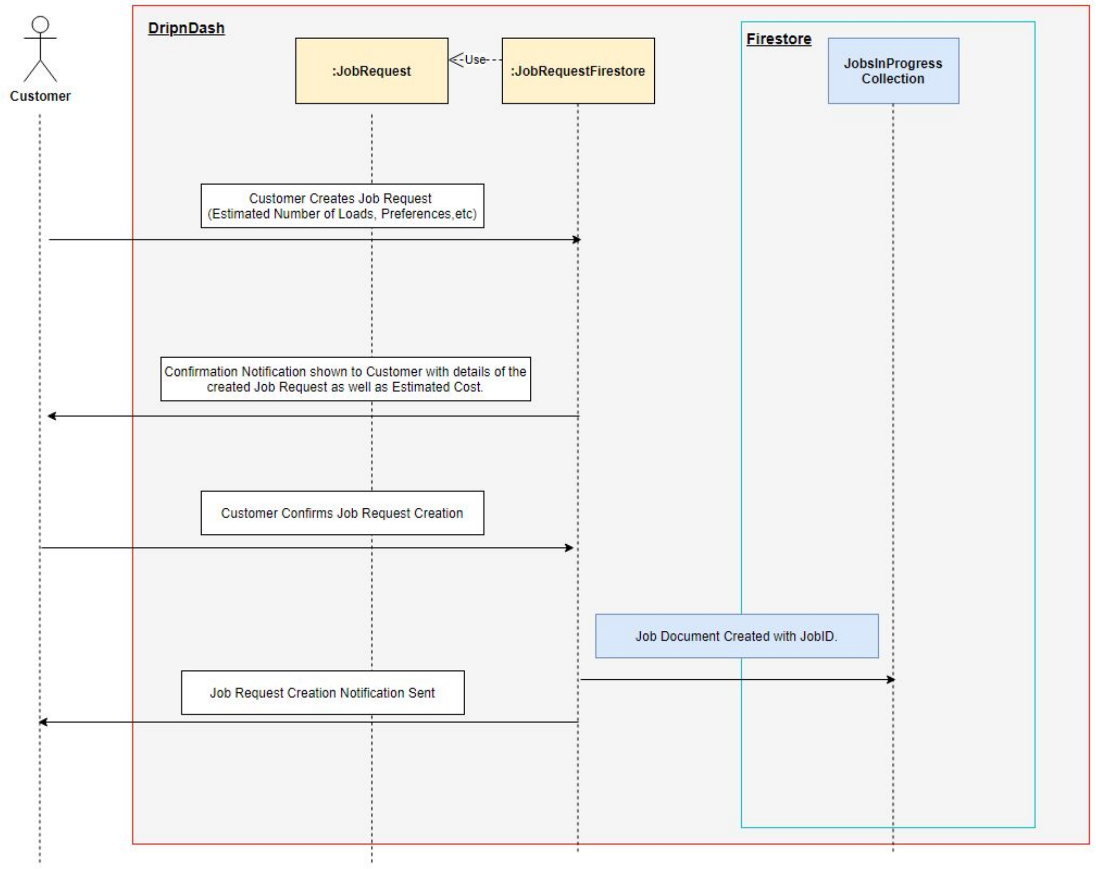
UC0.B: Customer/Dasher Login


DripnDash
Firestore
User
:SignInActivity
Collections
:CustomerHome
:DasherHomeActivity
User Enters Credentials
signInButton.setOnClickListener
if(user.getDisplayName().equals("dasher"))
"Logged In as a Dasher" Message shown to User
Dasher is sent to their HomeActivity Page
else
"Logged In as a Customer" Message shown to User
Customer is sent to their HomeActivity Page

<u>Design Principles</u>:

This figure shows the design sequence diagram for Use Case:Customer Login. If the user is registered in the system, one will be able to proceed to the Customer Home Page by typing customer's ID and passwords. The system will retrieve the stored credential and check if they are correct or incorrect. If it is correct, the user can log into the app.The design principle employed in this diagram is the Expert Doer Principle because there is a decent amount of communication in between the objects but it is short and focused. Therefore, since the communications are shortened between objects, this principle works best for this design.

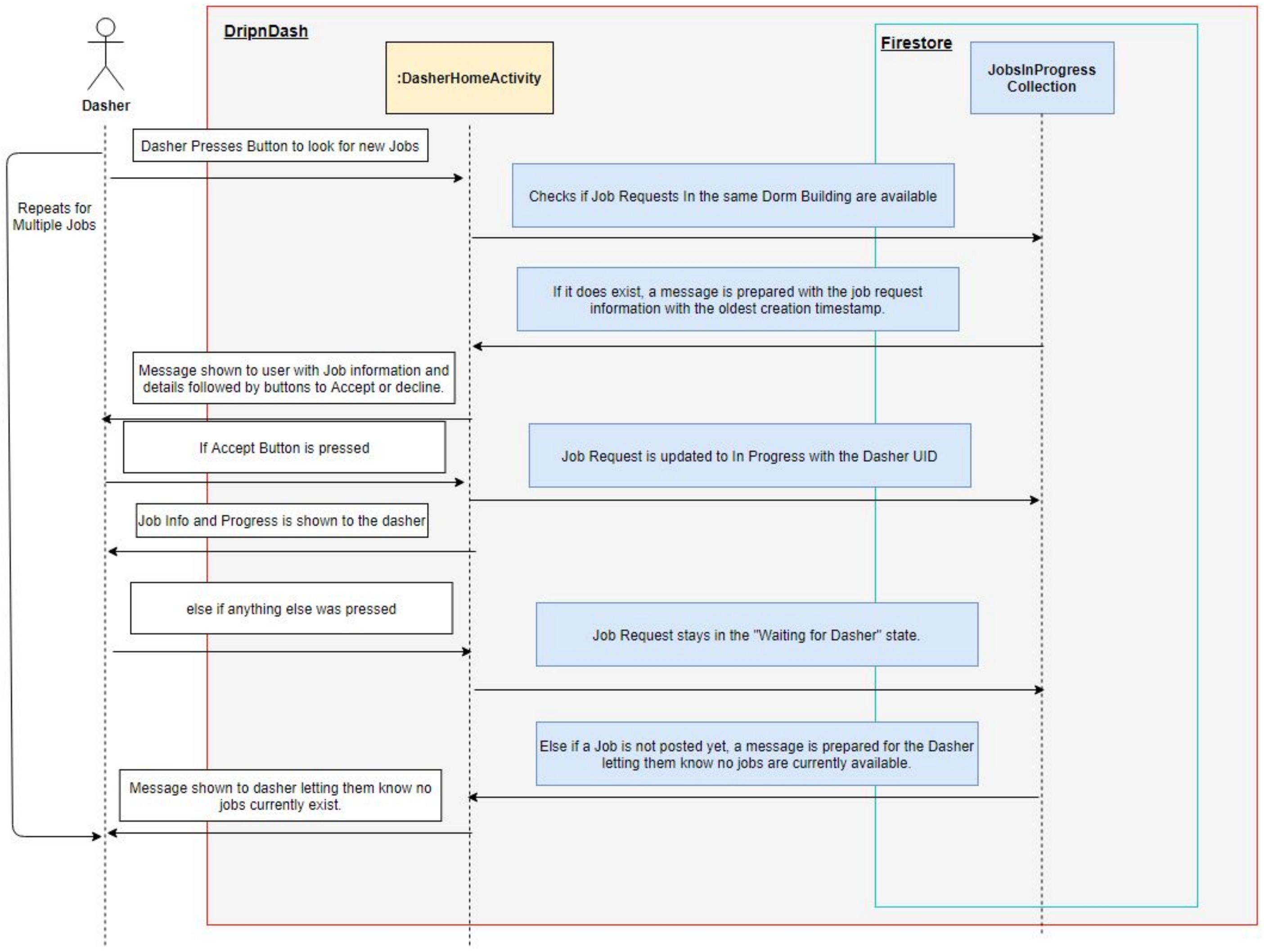
UC1: CustomerSubmitRequest


DripnDash
Customer
:JobRequest
Use
:JobRequestFirestore
Firestore
JobsInProgress Collection
Customer Creates Job Request (Estimated Number of Loads, Preferences,etc)
Confirmation Notification shown to Customer with details of the created Job Request as well as Estimated Cost.
Customer Confirms Job Request Creation
Job Document Created with JobID.
Job Request Creation Notification Sent

Design Principles:

This figure shows the design sequence diagram for Use Case: CustomerSubmitRequest. Once the customer is successfully logged in, the user can post a request for a laundry pickup by pressing the request button on the Customer Home Page. When the submission is completed, the server will store the customer's request information (Preferences, Number of Loads, etc). The stored data is added on the job list and the app will display the job status. The High Cohesion Principle is exemplified in this diagram since there is a lot of communication between the objects. All the data is being moved from location, to location to be stored. Therefore, there needs to be lots of communication.

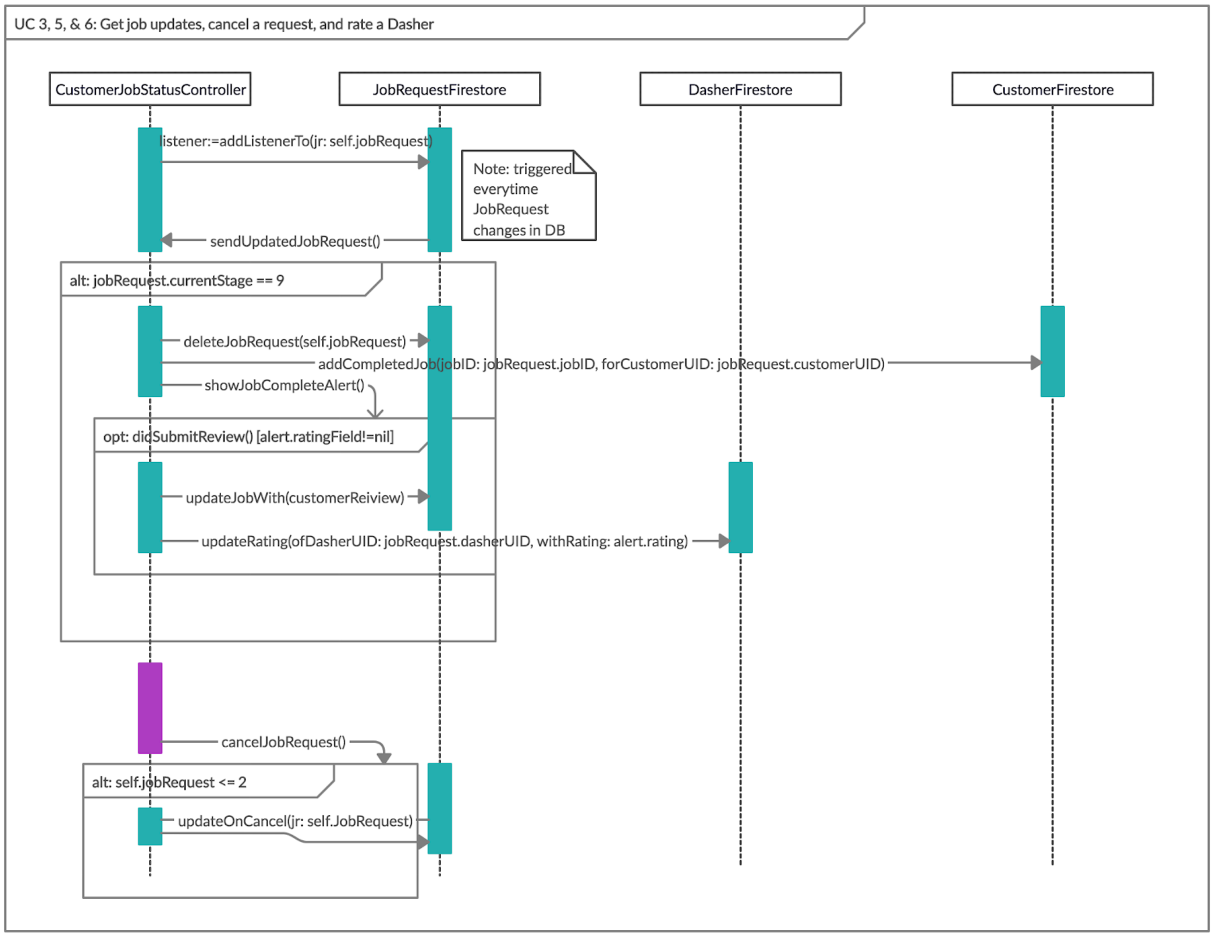
UC2: Dasher Accept Job


DripnDash
Firestore
Dasher
:DasherHomeActivity
JobsInProgress Collection
Repeats for Multiple Jobs
Dasher Presses Button to look for new Jobs
Checks if Job Requests In the same Dorm Building are available
If it does exist, a message is prepared with the job request information with the oldest creation timestamp.
Message shown to user with Job information and details followed by buttons to Accept or decline.
If Accept Button is pressed
Job Request is updated to In Progress with the Dasher UID
Job Info and Progress is shown to the dasher
else if anything else was pressed
Job Request stays in the "Waiting for Dasher" state.
Else if a Job is not posted yet, a message is prepared for the Dasher letting them know no jobs are currently available.
Message shown to dasher letting them know no jobs currently exist.

<u>Design Principles</u>:

This figure shows the sequence diagram for Use Cases: Dasher Accept Job. If the dasher is successfully logged into the application. The app will show available jobs on the list and the request will be added to Dasher's in progress job list when the dasher accepts a request. This method is favored by the low coupling principle and the diagram seems to be greatly simplified. However, the system should be responsible for all communication.

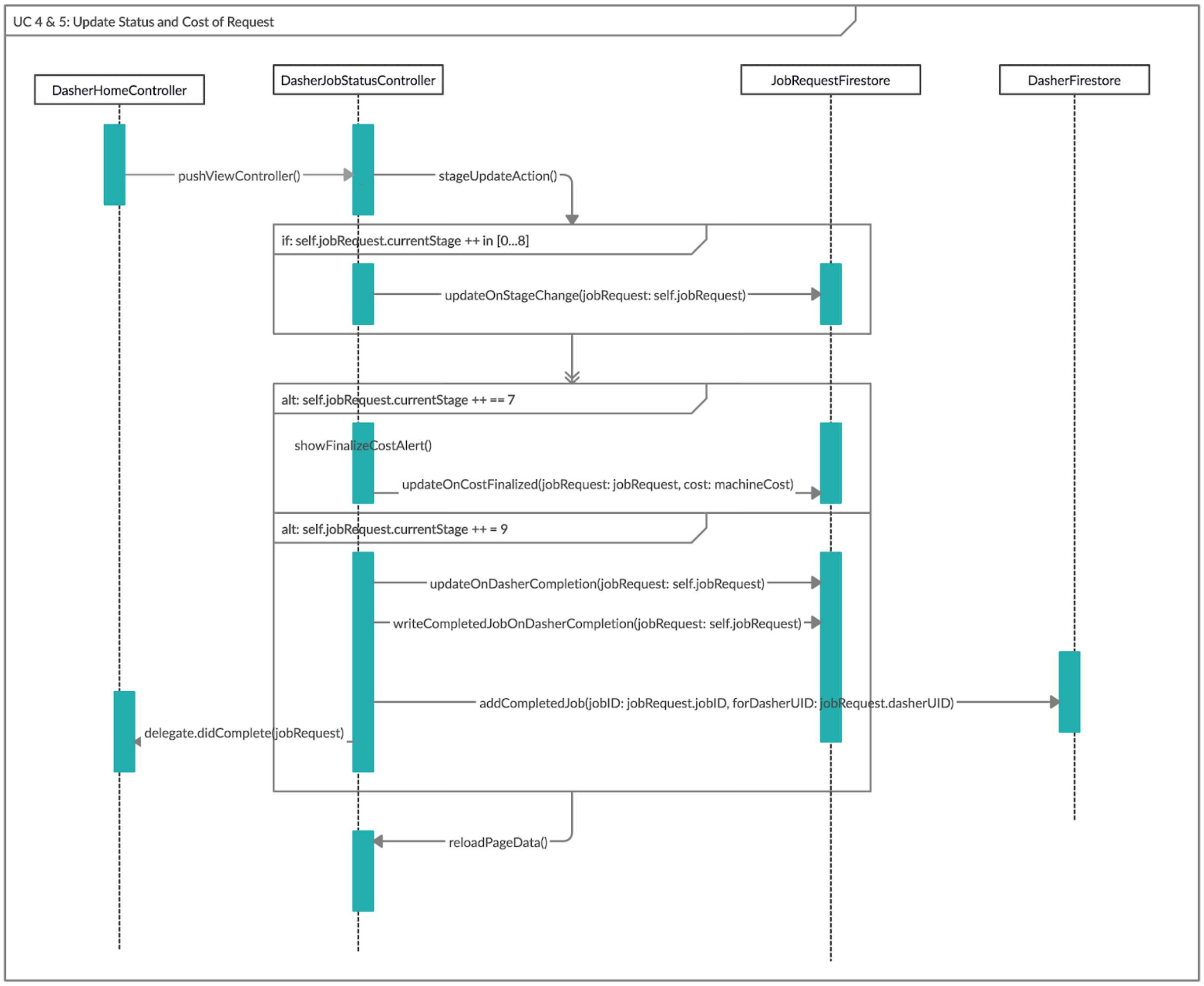
UC3, UC7 and UC6: See Status of Request, Customer: Cancel a Job Request and Review a Dasher


UC 3, 5, & 6: Get job updates, cancel a request, and rate a Dasher
CustomerJobStatusController
JobRequestFirestore
DasherFirestore
CustomerFirestore
listener:=addListenerTo(jr: self.jobRequest)
Note: triggered everytime JobRequest changes in DB
sendUpdatedJobRequest()
alt: jobRequest.currentStage == 9
deleteJobRequest(self.jobRequest)
addCompletedJob(jobID: jobRequest.jobID, forCustomerUID: jobRequest.customerUID)
showJobCompleteAlert()
opt: didSubmitReview() [alert.ratingField!=nil]
updateJobWith(customerReiview)
updateRating(ofDasherUID: jobRequest.dasherUID, withRating: alert.rating)
cancelJobRequest()
alt: self.jobRequest <= 2
updateOnCancel(jr: self.JobRequest)

Design Principles:

This figure shows the sequence diagram for Use Cases: See Status of Request, Update Cost Information of Request, Rate and Review a Dasher. In this, the Customer would be able to see the status of their requested job that is updated by the Dasher on their end. It also gives the Customer the ability to cancel the job they requested. Although, this is limited only to before the Dasher picks up the laundry. This design follows a High Cohesion Principle as the controller's responsibilities are limited to just coordinating the tasks between the different methods and nothing more.

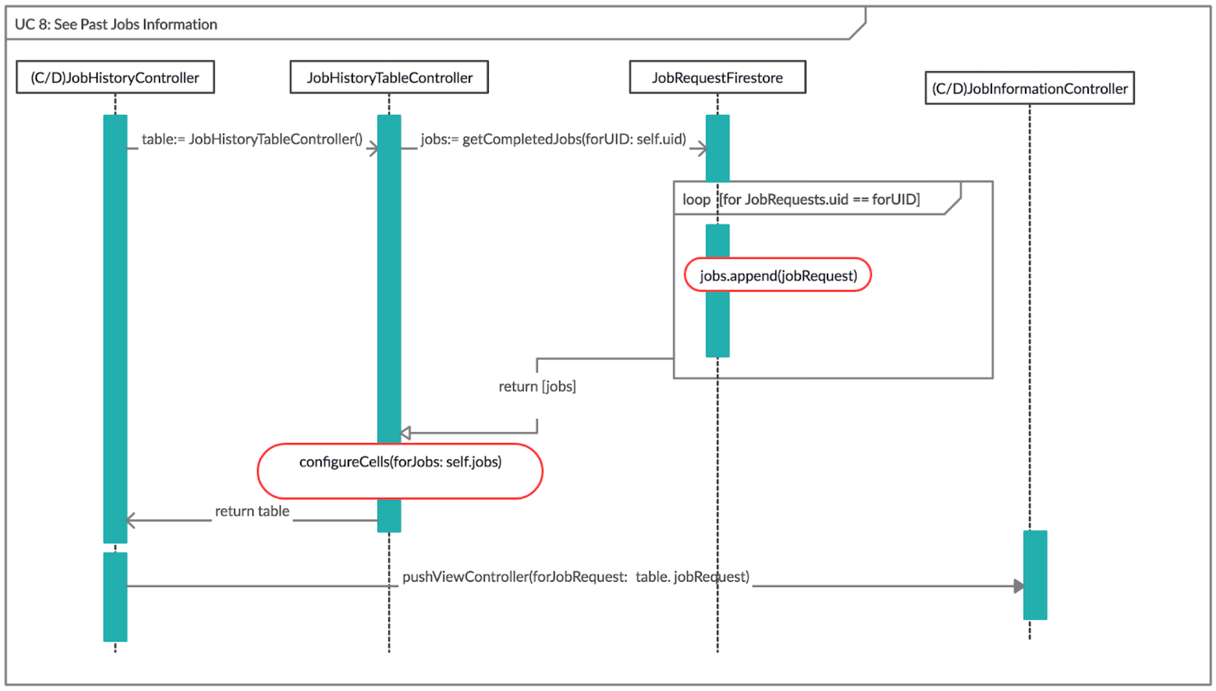
UC4 and UC5: Update Status of Request, Update Cost Information of Request


UC 4 & 5: Update Status and Cost of Request
DasherHomeController
DasherJobStatusController
JobRequestFirestore
DasherFirestore
pushViewController()
stageUpdateAction()
if: self.jobRequest.currentStage ++ in [0...8]
updateOnStageChange(jobRequest: self.jobRequest)
alt: self.jobRequest.currentStage ++ == 7
showFinalizeCostAlert()
updateOnCostFinalized(jobRequest: jobRequest, cost: machineCost)
alt: self.jobRequest.currentStage ++ = 9
updateOnDasherCompletion(jobRequest: self.jobRequest)
writeCompletedJobOnDasherCompletion(jobRequest: self.jobRequest)
addCompletedJob(jobID: jobRequest.jobID, forDasherUID: jobRequest.dasherUID)
delegate.didComplete(jobRequest)
reloadPageData()

<u>Design Principles</u>:

In this figure, it shows the Use Cases: Update Status of Request, Update Cost Information of Request. When the Customer requests a job, it displays the status of their job that the Dasher would continually update. Additionally, the cost of the request depending on the amount of loads can also get updated to the customer's end. We again employ the High Cohesion Principle so that the controllers only have the responsibility of making sure that the tasks are coordinated between the modules in these use cases.

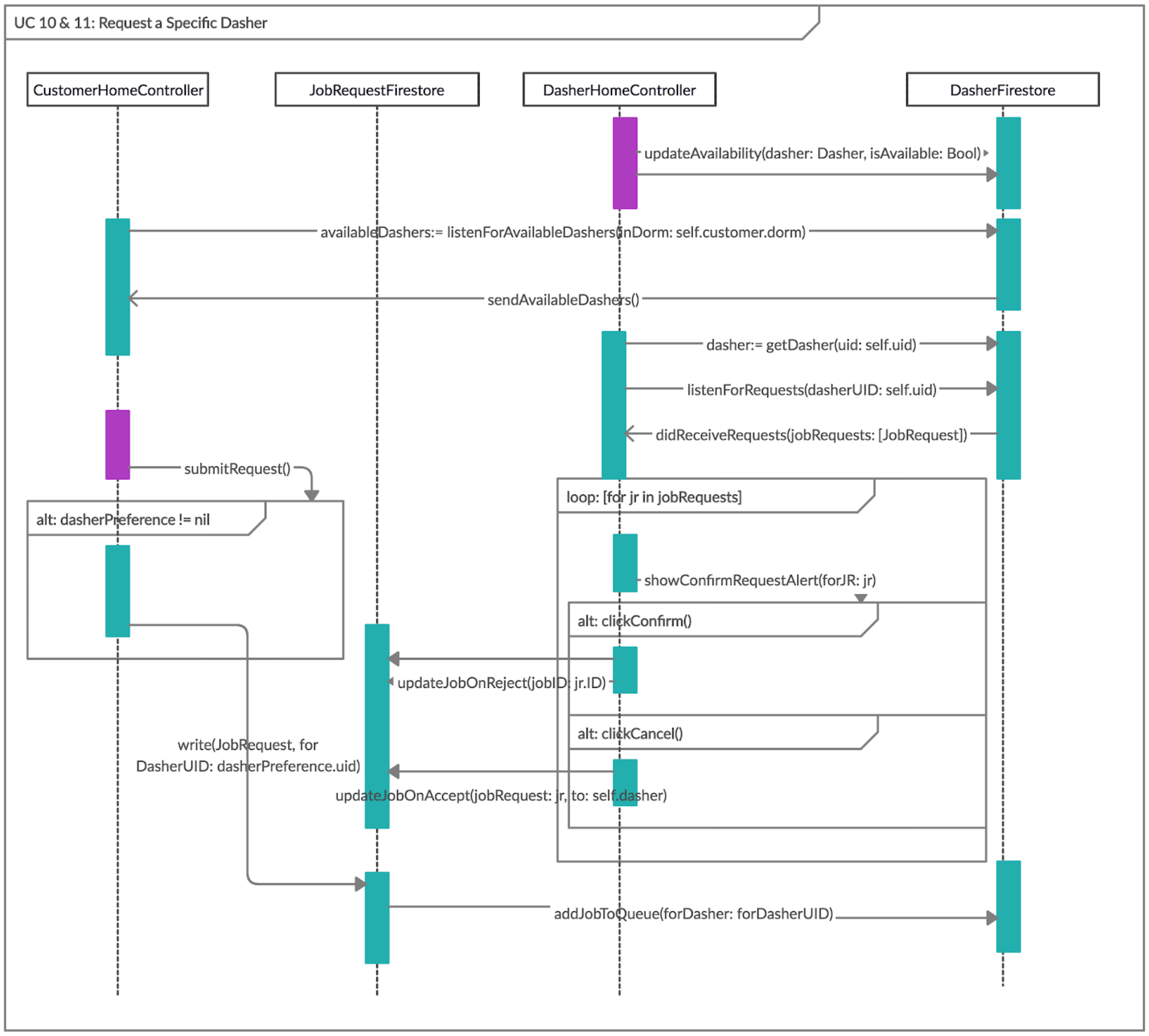
UC8: See information about Past Jobs


UC 8: See Past Jobs Information
(C/D)JobHistoryController
JobHistoryTableController
JobRequestFirestore
(C/D)JobInformationController
table:= JobHistoryTableController()
jobs:= getCompletedJobs(forUID: self.uid)
loop [for JobRequests.uid == forUID]
jobs.append(jobRequest)
return [jobs]
configureCells(forJobs: self.jobs)
return table
pushViewController(forJobRequest: table. jobRequest)

Design Principles:

In the figure presented, it shows the sequence diagram for the Use Case: See information about Past Job. The app gives the ability for Customers or Dashers to see their past jobs. It shows their past jobs in a list in which they would be able to see the details of said jobs. The controllers coordinate the information between each of them in order to fetch the previous jobs either the Customer requested of Dasher completed.

UC9 and UC10: Request a Specific Dasher, Change Request Acceptance Mode

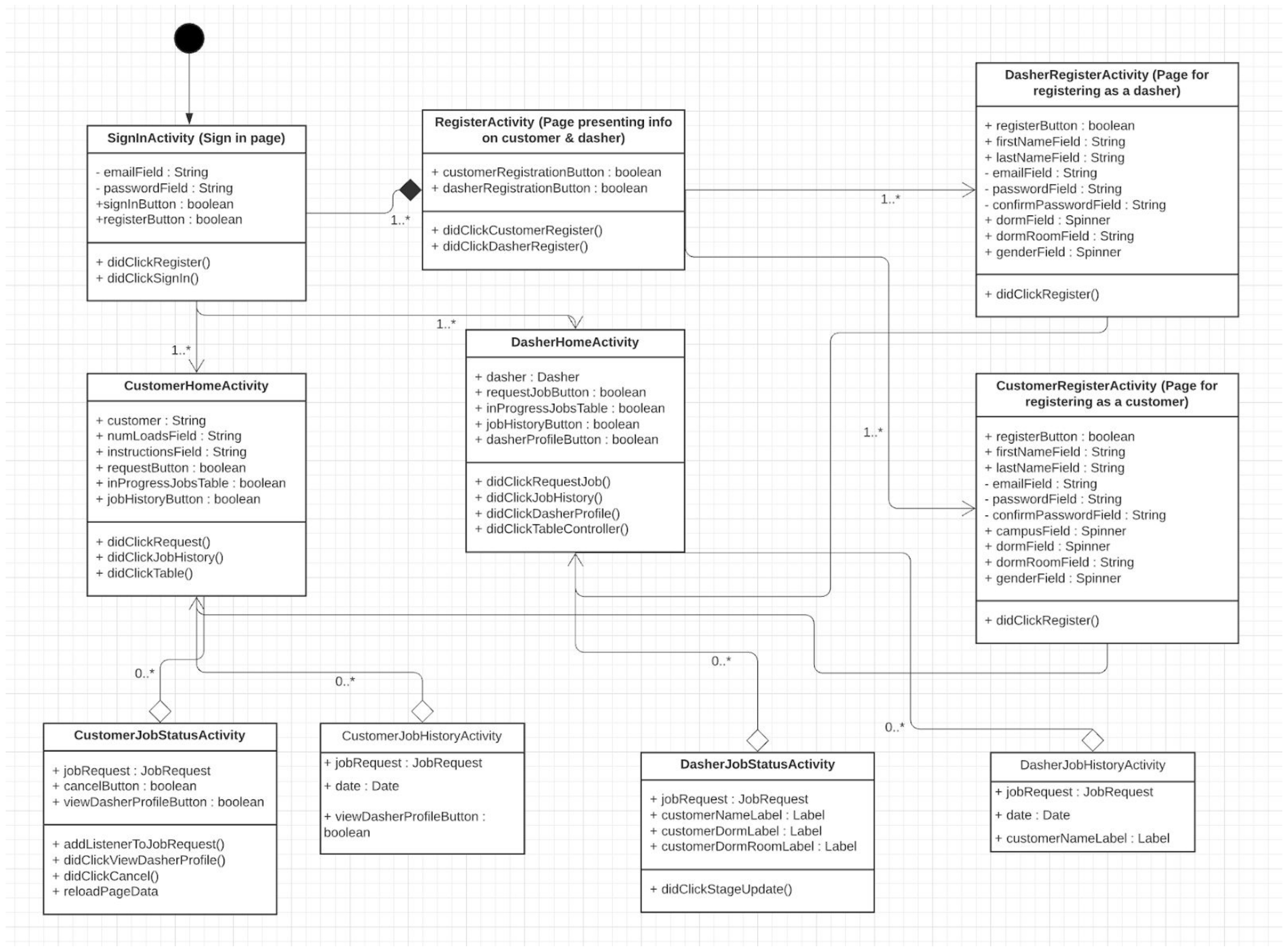

UC 10 & 11: Request a Specific Dasher
CustomerHomeController
JobRequestFirestore
DasherHomeController
DasherFirestore
updateAvailability(dasher: Dasher, isAvailable: Bool)
availableDashers:= listenForAvailableDashers(inDorm: self.customer.dorm)
sendAvailableDashers()
dasher:= getDasher(uid: self.uid)
listenForRequests(dasherUID: self.uid)
didReceiveRequests(jobRequests: [JobRequest])
submitRequest()
alt: dasherPreference != nil
loop: [for jr in jobRequests]
showConfirmRequestAlert(forJR: jr)
alt: clickConfirm()
updateJobOnReject(jobID: jr.ID)
alt: clickCancel()
write(JobRequest, for DasherUID: dasherPreference.uid)
updateJobOnAccept(jobRequest: jr, to: self.dasher)
addJobToQueue(forDasher: forDasherUID)

Design Principles:

In this design, it shows the Use Cases: Request a Specific Dasher, Change Request Acceptance Mode. From this, the Customer can request for a specific Dasher if they approved their work and would like to keep using their services. It also changes the Dashers Acceptance mode, which allows Dasher to either accept specific requests or accept any incoming requests. The design makes use of the High Cohesion principle, as each of the controllers only coordinate the tasks of the different modules.

Class Diagram and Interface Specification

Class Diagram



SignInActivity (Sign in page)
- emailField : String
- passwordField : String
+signInButton : boolean
+registerButton : boolean
+ didClickRegister()
+ didClickSignIn()
RegisterActivity (Page presenting info on customer & dasher)
+ customerRegistrationButton : boolean
+ dasherRegistrationButton : boolean
+ didClickCustomerRegister()
+ didClickDasherRegister()
DasherRegisterActivity (Page for registering as a dasher)
+ registerButton : boolean
+ firstNameField : String
+ lastNameField : String
- emailField : String
- passwordField : String
- confirmPasswordField : String
+ dormField : Spinner
+ dormRoomField : String
+ genderField : Spinner
+ didClickRegister()
CustomerHomeActivity
+ customer : String
+ numLoadsField : String
+ instructionsField : String
+ requestButton : boolean
+ inProgressJobsTable : boolean
+ jobHistoryButton : boolean
+ didClickRequest()
+ didClickJobHistory()
+ didClickTable()
DasherHomeActivity
+ dasher : Dasher
+ requestJobButton : boolean
+ inProgressJobsTable : boolean
+ jobHistoryButton : boolean
+ dasherProfileButton : boolean
+ didClickRequestJob()
+ didClickJobHistory()
+ didClickDasherProfile()
+ didClickTableController()
CustomerRegisterActivity (Page for registering as a customer)
+ registerButton : boolean
+ firstNameField : String
+ lastNameField : String
- emailField : String
- passwordField : String
- confirmPasswordField : String
+ campusField : Spinner
+ dormField : Spinner
+ dormRoomField : String
+ genderField : Spinner
+ didClickRegister()
CustomerJobStatusActivity
+ jobRequest : JobRequest
+ cancelButton : boolean
+ viewDasherProfileButton : boolean
+ addListenerToJobRequest()
+ didClickViewDasherProfile()
+ didClickCancel()
+ reloadPageData
CustomerJobHistoryActivity
+ jobRequest : JobRequest
+ date : Date
+ viewDasherProfileButton : boolean
DasherJobStatusActivity
+ jobRequest : JobRequest
+ customerNameLabel : Label
+ customerDormLabel : Label
+ customerDormRoomLabel : Label
+ didClickStageUpdate()
DasherJobHistoryActivity
+ jobRequest : JobRequest
+ date : Date
+ customerNameLabel : Label
1..*
1..*
1..*
1..*
1..*
0..*
0..*
0..*
0..*

Data Types and Operation Signatures

Activity Classes

- SignInActivity (Sign In Page)

  DEPENDENCIES: CustomerHomeActivity, DasherHomeActivity, RegisterActivity

  + emailField <EditText>
  + passwordField <EditText>
  + signInButton <Button onClick: “didClickSignIn”>

+ registerButton <Button onClick: “didClickRegister”>
# didClickRegister() {
  Description: Presents the RegisterActivity
  Calls: None
  Caller: self.registerButton ON(user clicks this button)
}
# didClickSignIn() {
  Description: Tries to sign the user in based on the information in emailField and passwordField. If the information is accurate, checks whether the user is a customer or dasher and presents the either CustomerHomeActivity or DasherHomeActivity
  Calls: None
  Caller: self.signInButton ON(user clicks this button)
}

- RegisterActivity (Page presenting info on Customer and Dasher)
  DEPENDENCIES: CustomerRegisterActivity, DasherRegisterActivity
  + customerRegisterButton <Button onClick: “didClickCustomerRegister”>
  + dasherRegisterButton <Button onClick: “didClickDasherRegister”>
  # didClickCustomerRegister() {
    Description: Presents the CustomerRegisterActivity
    Calls: None
    Caller: self.customerRegisterButton ON(user clicks this button)
  }
  # didClickDasherRegister() {
    Description: Presents the DasherRegisterActivity
    Calls: None
    Caller: self.dasherRegisterButton ON(user clicks this button)
  }
- CustomerRegisterActivity (Page for registering as a customer)
  DEPENDENCIES: Customer, CustomerFirestore
  + registerButton <Button onClick: “didClickRegister”>
  + firstNameField <EditText>
  + lastNameField <EditText>
  + emailField <EditText>
  + passwordField <EditText>
  + confirmPasswordField <EditText>
  + campusField <Spinner>

  + dormField <Spinner>
  + dormRoomField <EditText>
  + genderField <Spinner>
  # didClickRegister() {
    Description: Verifies the information the user submitted is valid to be a customer, then creates the user in Firebase and initializes the user’s data in the “customers” collection
    Calls: CustomerFirestore.isValidRegistration(), CustomerFirestore.initCustomerData(), Customer.init()
    Caller: self.registerButton ON(user clicks button)

}

- DasherRegisterActivity (Page for registering as a dasher)
  DEPENDENCIES: Dasher, DasherFirestore
  + registerButton <Button onClick: “didClickRegister”>
  + firstNameField <EditText>
  + lastNameField <EditText>
  + emailField <EditText>
  + passwordField <EditText>
  + confirmPasswordField <EditText>
  + dormField <EditText>
  + dormRoomField <EditText>
  + genderField <EditText>
  + ageField<EditText>
  # didClickRegister() {

    Description: Verifies the information the user submitted is valid to be a dasher, then creates the user in Firebase and initializes the user’s data in the “dashers” collection

    Calls: DasherFirestore.isValidRegistration(), DasherFirestore.initDasherData(), Dasher.init()

    Caller: self.registerButton ON(user clicks button)

  }
- CustomerHomeActivity (Page for customer home)
  DEPENDENCIES: Customer, CustomerFirestore, JobRequest, JobRequestFirestore, CustomerJobStatusActivity, CustomerJobHistoryActivity
  + customer: <Customer>
  + numLoadsField: <EditText>
  + instructionsField: <EditText>
  + requestButton: <Button onClick: “didClickRequest()“>
  + inProgressJobsTable: <TableLayout onClickItem: “didClickTable(item: JobRequest)“>
  + jobHistoryButton: <Button onClick: “didClickJobHistory”>
  # init() {

    Description: Set self.customer attribute

    Calls: CustomerFirestore.getCustomer()

    Caller: self

  }
  # didClickRequest() {

    Description: Creates a JobRequest object based on the information in numLoadsField, instructionsField, and self.customer and submits the request to Firestore. Then adds the JobRequest object to inProgressJobsTable.inProgressJobs and calls jobRequestFirestore.addListenerToJobRequest()

    Calls: jobRequest.init(), jobRequestFirestore.writeJobRequest(), jobRequestFirestore.addListenerToJobRequest()

    Caller: self.requestButton ON(user clicks request button after filling out request info)

  }
  # didClickJobHistory() {

    Description: Presents CustomerJobHistoryActivity

    Calls: None

Caller: self.jobHistoryButton ON(user clicks jobHistoryButton)
}
# didClickTable(item: JobRequest) {
Description: Presents the CustomerJobStatusActivity for the associated JobRequest the user clicked on in the table
Calls: None
Caller: self.inProgressJobsTable ON(user clicks an item in the table)
}

- DasherHomeActivity (Page for dasher home)
DEPENDENCIES: Dasher, DasherFirestore, JobRequest, JobRequestFirestore, DasherJobStatusActivity, DasherJobHistoryActivity
+ dasher: <Dasher>
+ requestJobButton: <Button onClick: "didClickRequestJob">
+ inProgressJobsTable: <TableLayout onClickItem: "didClickTableController(item: JobRequest)">
+ inProgressJobs: <[JobRequest]>
+ listeners: <[ListenerRegistration]>
+ jobHistoryButton: <Button onClick: "didClickJobHistory">
+ dasherProfileButton: <Button onClick: "didCickDasherProfile">
# init() {
Description: Set self.dasher attribute
Calls: DasherFirestore.getDasher()
Caller: self
}
# didClickRequestJob() {
Description: Gets the oldest JobRequest available to this dasher that is still pending assignment and assigns it to this dasher. Then adds this JobRequest object to self.inProgressJobs, and adds a listener to this object in self.listeners
Calls: JobRequestFirestore.getOldestJobRequest()
Caller: self.requestJobButton ON(user clicks requestJobButton)
}
# addListenerToJobRequest(jobRequest: JobRequest) {
Description: Appends a Firebase.ListenerRegistration to self.listeners for the argument jobRequest and defines the protocol for handling changes to the JR in the database. This listener only listens to the "WAS_CANCELLED" field and responds to this event. If this field is true, the associated JobRequest is removed from self.inProgressJobs and self.inProgressJobsTable, a notification of cancellation is presented to the user, and a
Calls: JobRequestFirestore.deleteJobRequest()
Caller: self ON(init)
}
# didClickJobHistory() {
Description: Presents DasherJobHistoryActivity
Calls: None
Caller: self.jobHistoryButton ON(user clicks jobHistoryButton)
}
# didClickDasherProfile() {
Description: Presents DasherProfileActivity

Calls: None
Caller: self.dasherProfileButton ON(user clicks this button)
}
# didClickTableController(item: JobRequest) {
Description: Presents the DasherJobStatusActivity for the associated JobRequest the user clicked on in the table
Calls: None
Caller: self.inProgressJobs ON(user clicks an item in the table)
}

- CustomerJobStatusActivity (page for customer to see info on in prog. req. (status, dasher etc.)

DEPENDENCIES: JobRequest, JobRequestFirestore

+ jobRequest: <JobRequest>
+ listener: <Firebase.ListenerRegistration>
+ cancelButton: <Button onClick: “didClickCancel()”>
+ viewDasherProfileButton: <Button onClick:”didClickViewDasherProfile()”>
# init(jobRequest: JobRequest) {
Description: Sets the value of self.jobRequest and self.listener
Calls: self.addListenerToJobRequest()
Caller: self
}
# addListenerToJobRequest(jobRequest: JobRequest) {
Description: Sets the value of self.listener as a Firebase.ListenerRegistration that listens to this JobRequest in Firstore. Also defines the protocol for updating the page data for when this value is changed in the database
Calls: self.reloadPageData()
Caller: self ON(init)
}
# didClickViewDasherProfile() {
Description: Presents the CustomerDasherProfileActivity for jobRequest.dasherUID
Calls: None
Caller: self.viewDasherProfileButton ON(user clicks this button)
}
# didClickCancel() {
Description: Removes the customer’s JobRequest from the database and notifies dasher that the job was cancelled. Note, the job can only be cancelled before the laundry has been picked up (before stage = 3)
Calls: JobRequestFirestore.updateOnCustomerCancel(),

Caller: self.cancelButton ON(user clicks this button)
}
# reloadPageData(forStage stage: Int) {
Description: If the currentStage data acquired by self.listener is in range [0,8] this
method
reloads the page data to reflect the current stage. Else (currentStage = 9) this method presents the CustomerReviewActivity
Calls: None
Caller: self.listener

}

- DasherJobStatusActivity (page fo dasher to see info on customer's req. (dormroom, customerName, etc.) and update current status of req.)
  DEPENDENCIES: JobRequest, JobRequestFirestore
  + jobRequest: <JobRequest>
  + customerNameLabel: <Label>
  + customerDormLabel: <Label>
  + customerDormRoomLabel: <Label>
  + stage2UpdateButton: <Button onClick: "didClickStageUpdate(forStage: 2)">
  + stage3UpdateButton: <Button onClick: "didClickStageUpdate(forStage: 3)">
  + stage4UpdateButton: <Button onClick: "didClickStageUpdate(forStage: 4)">
  + stage5UpdateButton: <Button onClick: "didClickStageUpdate(forStage: 5)">
  + stage6UpdateButton: <Button onClick: "didClickStageUpdate(forStage: 6)">
  + stage7UpdateButton: <Button onClick: "didClickStageUpdate(forStage: 7)">
  + stage8UpdateButton: <Button onClick: "didClickStageUpdate(forStage: 8)">
  + stage9UpdateButton: <Button onClick: "didClickStageUpdate(forStage: 9)">
  # didClickStageUpdate(forStage stage: Int) {

    Description: If stage is in range [2-8], this method updates self.jobRequest.currentStage and updates the jobRequest object in Firestore, and it updates this JobRequest object in DasherHomeActivity.inProgressJobsTable.inProgressJobs. Else (stage = 9), present JobCompletetionActivity

    Calls:

    Caller: stage(2-9)UpdateButton ON(user clicks nth update button)

    }

- CustomerDasherProfileActivity (page for customer to see dasher's profile. The dasher being the one who is doing the customer's laundry)
  DEPENDENCIES: Dasher, DasherFirestore
  + dasher: <Dasher>
  + dasherUID: <String>
  + dasherNameLabel: <Label>
  + dasherRatingLabel: <Label>
  + dasherNumJobsLabel: <Label>
  + dasherWorkLengthLabel: <Label>
  # init(dasherUID: String) {

    Description: Sets value of self.dasher using dasherUID, then populates all labels based on this dasher object

    Calls: DasherFirestore.getDasher()

    Caller: self

    }
- DasherProfileActivity (page for dasher to see/edit his/her own profile)
  DEPENDENCIES: Dasher, DasherFirestore
  + dasher: <Dasher>
  + nameLabel: <Label>
  + ratingLabel: <Label>
  + numJobsLabel: <Label>
  + workLengthLabel: <Label>

+ emailLabel: <Label>
# init(dasher: Dasher) {
    Description: Sets value of self.dasher, then populates all labels based on this dasher object
    Calls: None
    Caller: self
}

Database Management System Classes

- CustomerFirestore (Interface for reads/writes to Database/customers)
    + customersRef <CollectionReference> // reference to Database/customers
    # isValidRegistration(email: String, password: String, confirmPassword: String) -> Bool {
        Description: Checks if the information the user submitted is valid to be a customer
        Calls: None
        Callers: CustomerRegisterActivity ON(user presses register button after entering info)
    }

    # initCustomerData(customer: Customer) {
        Description: Write customer's data (see all fields in Customer obj) to Firestore and set user.displayName = "customer"
        Calls: None
        Callers: CustomerRegisterActivity ON(user presses register button after entering info)
    }

    # getCustomer(uid: String) -> Customer {
        Description: Read Database/customers/uid document and convert to Customer object
        Calls: None
        Callers: CustomerRegisterActivity ON(init)
    }

- DasherFirestore (Interface for reads/writes to Database/dashers)
    + dashersRef <CollectionReference> // reference to Database/dashers
    # isValidRegistration(email: String, password: String, confirmPassword: String, age: Int) -> Bool {
        Description: Checks if the information the user submitted is valid
        Calls: None
        Callers: DasherRegisterActivity ON(user presses register button after entering info)
    }
    # initDasherData(dasherr: Dasher) {
        Description: Write dasher's data to Firestore and set user.displayName = "customer"
        Calls: None
        Callers: DasherRegisterActivity ON(user presses register button after entering info)
    }
    # getDasher(uid: String) -> Dasher {
        Description: Read Database/dashers/uid document and convert to Dasher object
        Calls: None
        Callers: DasherHomeActivity ON(init)
    }

- JobRequestFirestore (Interface for reads/writes to JobRequest Objects in DB)
    + db = Firestore.firestore() // database link
    # writeJobRequest(jobRequest: JobRequest, isRewrite: Bool) {
        Description: Writes jobRequest to jobsPendingAssignment address and jobsInProgress address
        Calls: None
        Caller: CustomerHomeActivity ON(customer presses submit request), DasherHomeActivity ON(dasher denies job assignment)
    }
    # deleteJobRequest(jobRequest: JobRequest) {
        Description: Deletes the jobRequest object from Database/inProgressJobs address
        Calls: None
        Caller: DasherHomeActivity ON(user clicks confirm button on cancellation notification)
    }
    # updateOnAssignmentAccept(jobRequest: JobRequest) {
        Description: writes updated JR object to "jobInProgress/jobRequest.jobID"
        Calls: self.updateJobRequest()
        Caller: DasherHomeActivity ON(dasher press accept on job assignment notification)
    }
    # updateOnStageChange(jobRequest: JobRequest) {
        Description: writes updated JR object to "jobInProgress/jobRequest.jobID"
        Calls: self.updateJobRequest()
        Caller: DasherJobStatusActivity ON(dasher presses an update status button)
    }
    # updateJobRequest(jobRequest: JobRequest, fields: [AnyHashable: Any]) {
        Description: writes updated fields in JR object to "jobInProgress/jobRequest.jobID"
        Calls: None
        Caller: self.updateOnAssignmentAccept(), self.updateOnWorkerUpdate()
    }
    # getOldestJobRequest(availableToDasher dasher: Dasher) {
        Description: Gets documentID of oldest JR in queue and deletes the document. Then calls self.getInProgressJobRequest() to get the JR obj and send it to DasherHomeActivity
        Calls: self.getInProgressJobRequest()
        Caller: DasherHomeActivity ON(dasher press accept on job assignment notification)
    }
    # getInProgressJobRequest(fromDocumentID documentID: String, andAssignTo dasher: Dasher) {
        Description: Reads "jobsInProgress/documentID" and converts the information to a JR obj
        Caller: self.getOldestJobRequest
        Calls: None
    }
    # updateOnCustomerCancel(jobRequest: JobRequest) {
        Description: Sets "WAS_CANCELLED" field in "main" location to true
        Calls: self.updateJobRequest()
        Caller: CustomerJobStatusActivity ON(customer clicks cancel button)
    }

#   writeCompletedJobOnDasherCompletion(jobRequest: JobRequest) {
Description: Writes the jobREquest to jobsCompleted/jobID
Calls: None
Caller: DasherJobStatus
}

Data Structure Classes

- Customer
    - + firstName <String>
    - + lastName <String>
    - + email <String>
    - + dorm <String>
    - + dormRoom <String>
    - + gender <String>
    - + uid <String>
    - + age<int>
    - + completedJobs <[String]> // Array of jobIDs
- Dasher
    - + firstName <String>
    - + lastName <String>
    - + age <Int>
    - + email <String>
    - + dorm <String>
    - + dormRoom <Int>
    - + gender <String>
    - + uid <String>
    - + completedJobs <[String]> // Array of jobIDs
    - + rating <double> // num out of 5, initially set to 5
    - + numCompletedJobs <Int>
    - + registerTimestamp <Timestamp>
- JobRequest
    // assigned value on request creation (by customer)
    - + jobID <String>
    - + customerUID <String>
    - + requestTimestamp <Timestamp>
    - + customerName <String>
    - + dorm <String>
    - + dormRoom <String>
    - + customerInstructions <String>
    - + numLoadsEstimate <Int> // Customer's inputted estimate of number of loads
    // assigned value on assignment to dasher
    - + dasherUID <String>
    - + dasherName <String>

```
+    dasherRating <double>
+    assignedTimestamp <Timestamp>
     // dynamic properties (updated by dasher during job)
+    currentStage <Int>
     // properties assigned @ stage “finished drying” = stage7
+    numLoadsActual <Int>
+    machineCost <Double>
+    amountPaid <Double>
+    wasCancelled <Boolean>
+    completedTimestamp <timestmap>

     // properties for completed jobs
+    customerReview <String>
+    customerRating <Double>
     // static and computed properties
+    currentStatus <String>() -> self.stages.get(currentStage)
+    stages <Map<Int, String>> = {
             0: “Waiting for Dasher To Accept”, 1: “Dasher Accepted Request”,
             2: “Dasher on Way for Pickup”, 3: “Picked Up Laundry”,
             4: “Laundry in Washer”, 5: “Finished Washing“,
             6: “Laundry in Dryer“, 7: “Finished Drying“,
             8: “Dasher on Way for Drop Off“, 9: “Dropped Off Laundry“
     }
# updateOnAssignment(toDasher dasher: Dasher, time: Timestamp) {
     Description: Used to update self on assignment to Dasher
     Caller: DasherHomeActivity ON(user is assigned to a job)
     self.dasherUID = dasher.uid
     self.dasherName = dasher.firstName + “ “ + dasher.lastName
     self.dasherRating = dasher.rating
     self.assignedTimestamp = time
     self.currentStage = 1
     }
```

Traceability Matrix

Due to the amount of classes, we added a list of Domain Concepts and related classes instead of a visual representation

Domain Concepts:

Database:
Store all authentication information about registered customers and dashers, in progress laundry requests, master list of all completed laundry requests, etc.

Classes:

- CustomerFirestore - interface for data reads/writes to the database<->customers
- DasherFirestore - interface for data read/writes to the database<->customers
- JobRequestFirestore - Interface for reads/writes to JobRequest Objects in Database

SignIn:
Allow a registered user (customer/dasher) to sign in by entering his/her email and password, and provide link to register for the application

Classes:

- SignInActivity - Page on app that provides UI for users to sign-in
- RegisterActivity - Page on app that provides UI for users to register
- CustomerRegisterActivity - Specific page that provides UI for customers to register
- DasherRegisterActivity - Specific page that provides UI for dashers to register
- CustomerFirestore - Interface for reads/writes to database<->customers
- DasherFirestore - Interface for reads/writes to database<->customers

RegisterInformation:
Provide information to an unregistered user what a customer is and what a dasher is in our application.

Classes:

- SignInActivity - Page on app that provides UI for users to sign-in
- RegisterActivity - Page on app that provides UI for users to register
- CustomerRegisterActivity - Specific page that provides UI for customers to register
- DasherRegisterActivity - Specific page that provides UI for dashers to register
- CustomerFirestore - Interface for reads/writes to database<->customers
- DasherFirestore - Interface for reads/writes to database<->customers

CustomerRegister:
Allow an unregistered user to enter his/her information and sign up for our application as a customer.

Classes:
- SignInActivity - Page on app that provides UI for users to sign-in
- RegisterActivity - Page on app that provides UI for users to register
- CustomerRegisterActivity - Specific page that provides UI for customers to register
- CustomerFirestore - Interface for reads/writes to database<->customers

DasherRegister:
Allow an unregistered user to enter his/her information and sign up for our application as a dasher.

Classes:
- SignInActivity - Page on app that provides UI for users to sign-in
- RegisterActivity - Page on app that provides UI for users to register
- CustomerRegisterActivity - Specific page that provides UI for customers to register
- DasherRegisterActivity - Specific page that provides UI for dashers to register
- CustomerFirestore - Interface for reads/writes to database<->customers
- DasherFirestore - Interface for reads/writes to database<->customers

CustomerFirestore:
Determine if the information a user entered to register as a customer is valid, and initialize customer's data in the database (Firestore). Interface for the application and reads/writes to customer's information in the database

Classes:
- CustomerRegisterActivity - Specific page that provides UI for customers to register
- CustomerFirestore - Interface for reads/writes to database<->customers
- RegisterActivity - Page on app that provides UI for users to register

DasherFirestore:
Determine if the information a user entered to register as a dasher is valid, and initialize customer's data in the database (Firestore). Interface for the application and reads/writes to dasher's information in the database

Classes:
- RegisterActivity - Page on app that provides UI for users to register
- DasherRegisterActivity - Specific page that provides UI for dashers to register
- DasherFirestore - Interface for reads/writes to database<->customers

Customer:
Container for customer's data frequently used by other parts of the system (firstName, lastName, email, dorm, dormRoom, etc.)

Classes:

- SignInActivity - Page on app that provides UI for users to sign-in
- CustomerHomeActivity - page for customer home
- CustomerJobStatusActivity - page for customer to see info on in prog. req. (status, dasher etc.)
- CustomerDasherProfileActivity - page for dasher to see info on customer's req.
- CustomerJobHistoryActivity - page for customer to see info on all his/her past requests
- CustomerJobInformationActivity - page for customer to see all information on a past job request
- CustomerFirestore - Interface for reads/writes to Database/customers

Dasher:
Container for dasher's data frequently used by other parts of the system (firstName, lastName, email, dorm, dormRoom, etc.)

Classes:

- SignInActivity - Page on app that provides UI for users to sign-in
- DasherHomeActivity - Page for dasher home
- DasherJobStatusActivity - page for dasher to see info on customer's req. (dorm room, customerName, etc.) and update the current status of req.
- CustomerDasherProfileActivity - page for customers to see dasher's profile. The dasher being the one who is doing the customer's laundry
- DasherJobHistoryActivity - page for dasher to see info on all his/her past requests
- DasherJobInformationActivity - page for dasher to see all information on a past job
- DasherFirestore - Interface for reads/writes to Database/dashers

DasherHome:
Allow a dasher to accept new jobs, and render a list with basic information about the current status of the dasher's in progress jobs.

Classes:

- Dasher - Data structure class for dasher
- DasherFirestore - Interface for reads/writes to Database/dashers
- JobRequest - data structure for jobs that were requested
- JobRequestFirestore - Interface for reads/writes to JobRequest Objects in DB
- DasherJobStatusActivity - page for dasher to see info on customer's req. (dorm room, customerName, etc.) and update the current status of req.
- DasherJobHistoryActivity - page for dasher to see info on all his/her past requests

CustomerHome:
Allow a customer to enter information about his/her request for laundry and submit it, and render an interactive list with all of a customer's in progress jobs

Classes:
- Customer - Data structure for customers
- CustomerFirestore - Interface for reads/writes to Database/customers
- JobRequest - data structure for jobs that were requested
- JobRequestFirestore - Interface for reads/writes to JobRequest Objects in DB
- CustomerJobStatusActivity - page for customer to see info on in prog. req. (status, dasher etc.)
- CustomerJobHistoryActivity - page for customer to see info on all his/her past requests

CustomerJobStatus:
Present detailed information about the current status of a customer's in progress laundry request including (CURRENT_STAGE in [“pending dasher assignment”, “dasher outside for pickup”, etc.], estimated drop off time, link to DasherProfile of dasher assigned to the request, etc.)

Classes:
- CustomerJobStatusActivity - page for customer to see info on in prog. req. (status, dasher etc.)
- Customer - Data structure for customers
- CustomerFirestore - Interface for reads/writes to Database/customers

CustomerJobHistory:
Present an interactive list of all a customer's completed requests for laundry with basic information about each request

Classes:
- CompletedJob - data structure for completed jobs
- CustomerFirestore - interface for data reads/writes to the database<->customers

CustomerPastJobInfo:
Present detailed information about a customer's past laundry request including (timeStamp the request was submitted, timeStamp the request was completed, dasher that completed the request, amount paid, etc.)

Classes:
- CompletedJob - data structure for completed jobs
- CustomerFirestore -interface for data reads/writes to the database<->customers

DasherJobStatus:
Allow a dasher to update the current status of a customer's laundry request (CURRENT_STATUS in [“Outside for Pickup”, “Laundry in Washer”, “Outside for Drop Off”, etc.]

Classes:

- DasherFirestore - interface for data read/writes to the database<->customers
- JobRequest - data structure for jobs that were requested
- JobRequestFirestore - Interface for reads/writes to JobRequest Objects in DB
- DasherJobStatusActivity - (page for dasher to see info on customer's req. (dorm room, customerName, etc.) and update the current status of req.
- DasherJobHistoryActivity - page for dasher to see info on all his/her past requests

DasherJobHistory:
Present an interactive list of all a dasher's completed jobs with basic information about each job

Classes:

- CompletedJob - data structure for completed jobs
- Dasher - data structure for dashers
- DasherFirestore - interface for data read/writes to the database<->customers

JobRequest:

Container for all information related to a customer's job/laundry request including (jobID, number of loads, customer's dorm, customer's dorm room, etc.)

Classes:

- JobRequestFirestore - Interface for reads/writes to JobRequest Objects in DB
- CustomerFirestore - interface for data reads/writes to the database<->customers
- DasherFirestore - interface for data read/writes to the database<->customers

jobRequestFirestore:

Handle reads/writes about JobRequest information to the database (Firestore). Interface for the application and the JobRequest portion of the database.

Classes:

- CustomerFirestore - interface for data reads/writes to the database<->customers
- DasherFirestore - interface for data read/writes to the database<->customers
- JobRequestFirestore -Interface for reads/writes to JobRequest Objects in DB

System Architecture and System Design

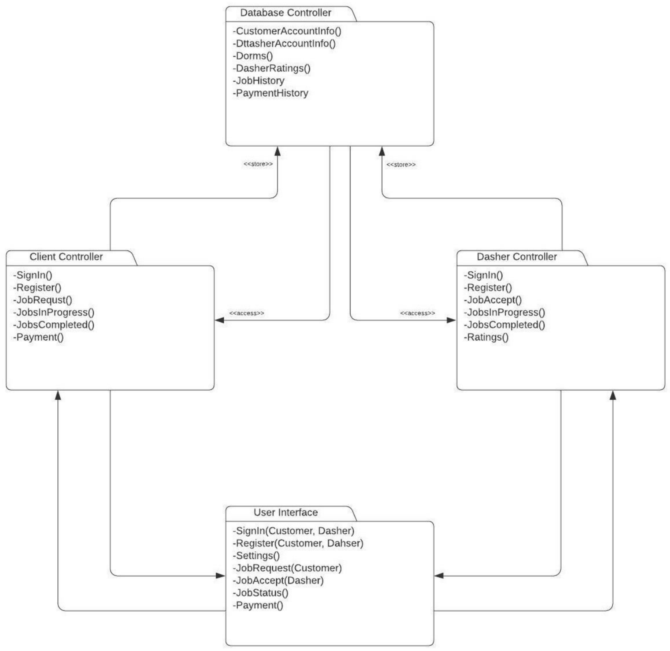
Architectural Styles

One key architecture that is being implemented is the use of Pipe-and-Filter architecture, where there is a flow of events that occur once an action is done - an example would be how the dasher updates the status of the job they are assigned to. For starters, the dasher would press a button on the UI indicating the progress of the job being updated. Then, the app on the dasher's end would send that data to the server for a notification to be prepared for display on the customer's end. The data would be sent from the dasher to the server to the user with the notification, a progress bar update on both ends, as well as a completion time estimation, all because of an initial action. This would be the case as well with the initial job acceptance and the drop off notification as well as the in app messaging system. This is interchanged with an Event-driven Architecture (EDA). because most of the stages of a job is progressed through a user's action - an event - in order to visually notify the customer about the stage of their job request.

Since Databases do play an important role in this project, as one cannot even access the features of the app without logging in, we also planned on implementing a Database-centric architecture in terms of credentials and reviews. Whenever the user logs in for the first time, they are required to create credentials to log into the application, those of which are saved in the database. In addition, the requested credentials would also be stored in the database. The reviews in the database would be updated whenever a dasher completes a job, after which the customer reviews the dasher and vice versa.

Identifying Subsystem

There are four major subsystems. The first being the database controller, the client controller, the dasher controller, and the user interface controller. The database controller will be dependent on both the dasher and client controller which will be dependent on the user interface controller. The user interface controller lets the user decide what operation they would like to do. Depending on who is using the application whether it be the client or dasher the user interface controller along with each different controller will communicate with the database via the database controller. The database controller will tell the server what information should be written to the database or should be sent from the database. Once the information is received depending on who the user is there respective controller either client or dasher will present the options or uses to the user interface subsystem so that the user can then continue to use the application.



Mapping Subsystems to Hardware

The system should be able to run on most mobile devices that have an internet connection.. The majority of the functionality would be provided via the internet using Firebase. Therefore, as long as the device is able to contact the Firebase server the application will run smoothly.

Persistent Data Storage

The system requires data to be saved in order to outlive a single execution of the system. The data includes user account information, log data of job status, payment history and dasher's review ratings. Firebase, a cloud-based database, will be used to store the data collected from the system. Firebase uses NoSQL, a non-relational database. SQL databases are known as relational databases, and have a table-based data structure, with predefined schema required. On the other hand, NoSQL databases do not need a predefined schema which allows us to work more freely with unstructured data.

Database Schema Note: (C) -> Collection, (D) -> Document, (F) -> Field

- customers (C)
    - uid (D)
        - UID: String (F)
        - FIRST_NAME: String (F)
        - LAST_NAME: String (F)
        - EMAIL: String (F)
        - DORM: String (F)
        - DORM_ROOM: Int (F)
        - GENDER: String (F)
        - COMPLETED_JOBS: [String] (F)
- dashers (C)
    - uid (D)
        - UID: String (F)
        - FIRST_NAME: String (F)
        - LAST_NAME: String (F)
        - EMAIL: String (F)
        - DORM: String (F)
        - DORM_ROOM: Int (F)
        - GENDER: String (F)
        - AGE: Int (F)
        - COMPLETED_JOBS: [String] (F)
        - RATING: Double (F)
        - NUM_COMPLTED_JOBS: Int (F)
        - REGISTER_TIMESTAMP: Timestamp (F)
        - REQUESTS_PENDING_ACCEPT: [String] (F)
- dorms (C)
    - dormName (D)
        - availableDashers: {uid: String, name: String, rating: Double} (F)
        - jobsPendingAssignment (C)
            - jobID (D)
                - REQUEST_TIMESTAMP: Timestamp (F)
- jobsInProgress (C)
    - jobID (D)
        - REQUEST_TIMESTAMP: Timestamp (F)

        - ASSIGNED_TIMESTAMP: Timestamp (F)
        - COMPLETED_TIMESTAMP: Timestamp (F)
        - CUSTOMER_NAME: String (F)
        - CUSTOMER_INSTRUCTIONS: String (F)
        - CUSTOMER_UID: String (F)
        - NUM_LOADS_ESTIMATE: Int (F)
        - DORM: String (F)
        - DORM_ROOM: String (F)
        - DASHER_UID: String (F)
        - DASHER_NAME: String (F)
        - DASHER_RATING: Double (F)
        - CURRENT_STAGE: Int (F)
        - NUM_LOADS_ACTUAL: Int (F)
        - MACHINE_COST: Double (F)
        - AMOUNT_PAID: Double (F)
        - WAS_CANCELLED: Boolean (F)
        - WAS_REJECTED: Boolean (F)

- jobsCompleted (C)
    - jobID (D)
        - REQUEST_TIMESTAMP: Timestamp (F)
        - ASSIGNED_TIMESTAMP: Timestamp (F)
        - COMPLETED_TIMESTAMP: Timestamp (F)
        - CUSTOMER_NAME: String (F)
        - CUSTOMER_INSTRUCTIONS: String (F)
        - CUSTOMER_UID: String (F)
        - NUM_LOADS_ESTIMATE: Int (F)
        - DORM: String (F)
        - DORM_ROOM: Int (F)
        - DASHER_UID: String (F)
        - DASHER_NAME: String (F)
        - DASHER_RATING: Double (F)
        - NUM_LOADS_ACTUAL: Int (F)
        - MACHINE_COST: Double (F)
        - AMOUNT_PAID: Double (F)
        - CUSTOMER_RATING: Double (F)
        - CUSTOMER_REVIEW: String (F)

Network Protocol

When it comes to the network protocol, we are using the Firebase Firestore Database so that all of the customers can automatically send and receive data in the application. Firebase uses JSON to store data and it is synchronized in real time to every connected client. In case of clients communicating with the database, they will be connected to the database and will maintain an open bidirectional connection via websocket. Then, if any client pushes data to the database it will be triggered and inform all connected clients that it has been changed by sending them the newly saved data. The benefit of using the Firebase is, for our system, data must be persisted and even when it goes offline and regains connectivity, the database synchronizes the local data changes with the remote update that occured while the client was offline[5].

Global Control Flow

Execution Orderliness: Our main events are requests. The system waits for requests from Dashers and customers. Dashers request for jobs while customers request for Dashers. Without any requests, the system will remain idle, waiting for events.

Time Dependency: There are no timers in the system. However, the system does run in real-time in response to events. The timestamp of an event is taken into account since events with the oldest timestamps are generally responded to before events with newer timestamps.

Concurrency: Each dorm is its own self-contained system and the events and jobs in one dorm do not affect the events in another. In that regard, each dorm could be considered a thread. Within each dorm, each user using the app is considered a separate thread once they submit a request for a job or for a Dasher. The initial requests are not synchronized at all. However, when a Dasher is paired with a customer, the events that take place on their phones become synchronized. For example, if a Dasher accepts the customer's job, the customer will receive a notification about that and the customer will continue to receive updates from the Dasher. Once the job is completed and the customer submits their review, the two threads of the Dasher and customer are not synchronized anymore.

Hardware Requirements

The bare minimum requirement to use this app is a smartphone that was made to be able to support the latest software versions on both Android and iOS devices. Most apps are downloaded from the respective store on their device, so a minor amount of storage space is required. A reliable communication signal is pivotal to the usage of this app because, without it, the dasher and the customer would not be able to communicate with each other, resulting in a negative experience on both ends.

Algorithms and Data Structure

Algorithms

Dasher Requests to be Assigned a Job

- Get all JobRequest documents from database address "dorms/Dasher.dormName/jobsPendingAssignment"
- Sort documents by field "REQUEST_TIMESTAMP" and get documentID of oldest document, delete this document, and return the documentID
- Read address "jobsInProgress/documentID" and convert data to a JobRequest object and return it to DasherHomePage for the requesting dasher

Data Structures

- Array: Arrays are used to store job requests. We used arrays because they are used in populating table views. The table views store cards of information about each job. Also, it is easy to remove and add items to the array, making it easy to add and remove items from the table view.

User Interface Design and Implementation

Unmodified (fully developed) Screen Mock-ups

- SignInPage, RegisterPage, CustomerRegisterPage, DasherRegisterPage

  The pages listed above remain unchanged in our application, aside from the addition of a register button to the SignInPage.

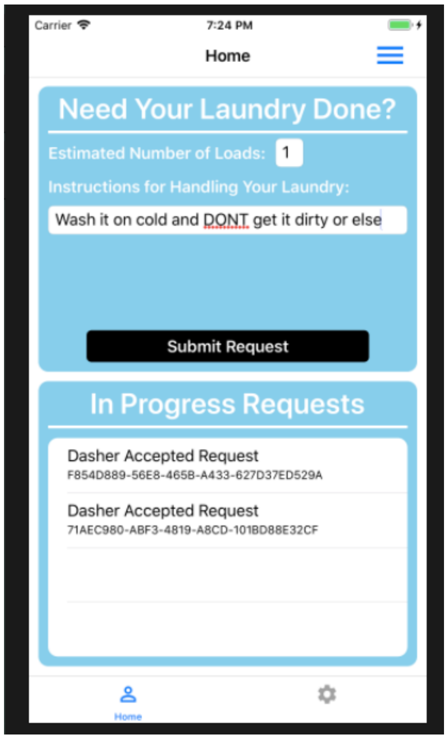
Unmodified (still undeveloped) Screen Mock-ups

- CustomerJobsHistoryPage, DasherJobsHistoryPage, CustomerJobInformationPage, DasherJobInformationPage

The pages listed above remain unchanged in terms of their function and design, but are still undeveloped.

iOS User Interface

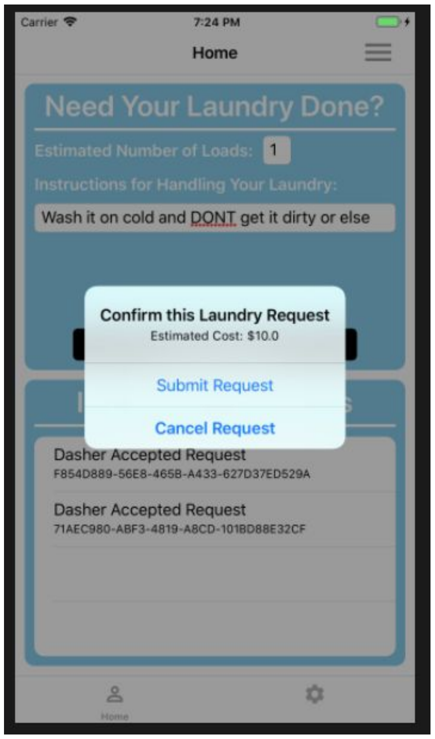
- **CustomerHomePage**
  - This page was modified to allow a user logged in as a customer to perform two core functions: submitting a request for a dasher to pick up laundry, and to see a table with all the customer's in progress jobs.



Wash it on cold and DONT get it dirty or else

Dasher Accepted Request
F854D889-56E8-465B-A433-627D37ED529A

Dasher Accepted Request
71AEC980-ABF3-4819-A8CD-101BD88E32CF



Wash it on cold and DONT get it dirty or else

Confirm this Laundry Request
Estimated Cost: $10.0

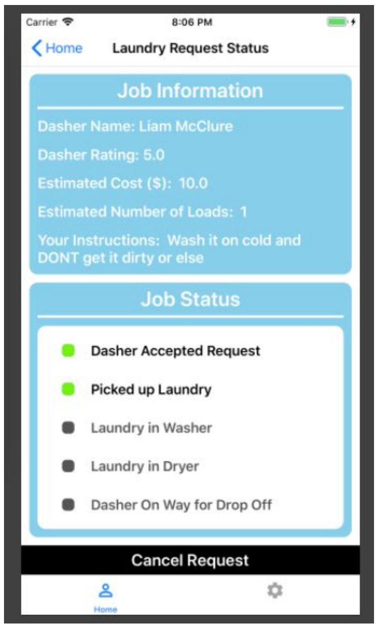
Dasher Accepted Request
F854D889-56E8-465B-A433-627D37ED529A

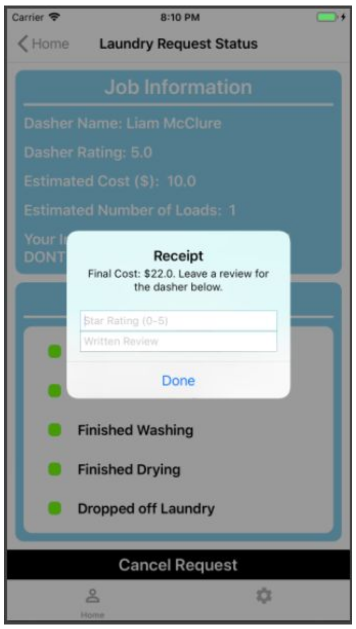
Dasher Accepted Request
71AEC980-ABF3-4819-A8CD-101BD88E32CF

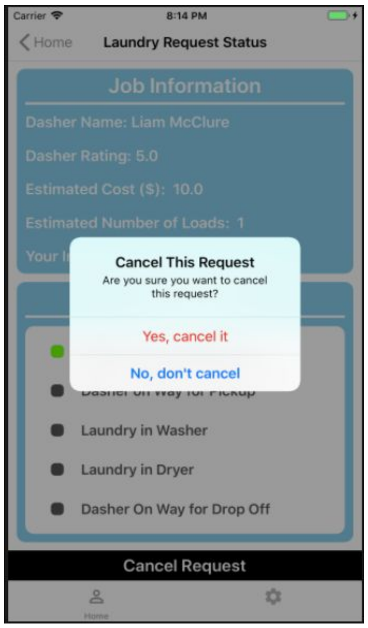
  - The **image to the left** depicts the homepage a user sees after successfully logging in as a customer. The **top portion** is where the user can enter the estimated number of loads the laundry contains, and any instructions the dasher should follow when doing laundry. At the bottom of the "Need Your Laundry Done?" section is the button the customer clicks in order to submit the laundry request. Clicking this button prompts the pop up in the **image on the right.** This pop-up displays the estimated cost of the request, and allows the user to confirm or cancel it. If the user clicks the "Submit Request" button, the job laundry request will be posted (written to the database). Otherwise, clicking the "Cancel Request" button will not post the request.
  - The **bottom portion** of the **image on the left** is where the customer can see all his or her in progress requests. Currently, each item in the table displays the status of

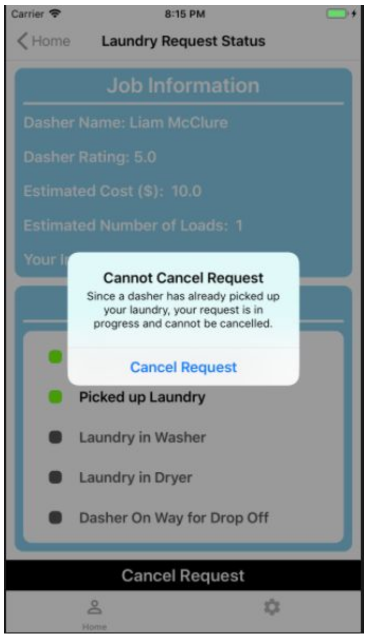
the request and the jobID of the request. Clicking on an item in this table will present the **CustomerJobStatusPage**, which is defined below.

- **CustomerJobStatusPage**
  - This page will present the customer with information about the dasher assigned to the request, basic cost information about the request, and the current status of the request


Carrier 8:06 PM
Home Laundry Request Status
Job Information
Dasher Name: Liam McClure
Dasher Rating: 5.0
Estimated Cost ($): 10.0
Estimated Number of Loads: 1
Your Instructions: Wash it on cold and DONT get it dirty or else
Job Status
🟩 Dasher Accepted Request
🟩 Picked up Laundry
● Laundry in Washer
● Laundry in Dryer
● Dasher On Way for Drop Off
Cancel Request
Home


Carrier 8:10 PM
Home Laundry Request Status
Job Information
Dasher Name: Liam McClure
Dasher Rating: 5.0
Estimated Cost ($): 10.0
Estimated Number of Loads: 1
Receipt
Final Cost: $22.0. Leave a review for the dasher below.
Star Rating (0-5)
Written Review
Done
🟩 Finished Washing
🟩 Finished Drying
🟩 Dropped off Laundry
Cancel Request
Home

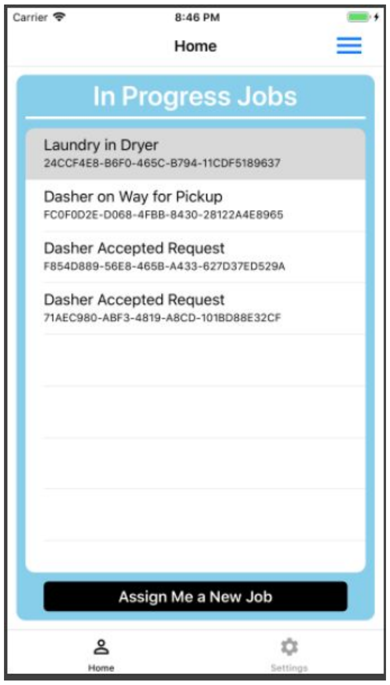

Carrier 8:14 PM
Home Laundry Request Status
Job Information
Dasher Name: Liam McClure
Dasher Rating: 5.0
Estimated Cost ($): 10.0
Estimated Number of Loads: 1
Cancel This Request
Are you sure you want to cancel this request?
Yes, cancel it
No, don't cancel
● Laundry in Washer
● Laundry in Dryer
● Dasher On Way for Drop Off
Cancel Request
Home


Carrier 8:15 PM
Home Laundry Request Status
Job Information
Dasher Name: Liam McClure
Dasher Rating: 5.0
Estimated Cost ($): 10.0
Estimated Number of Loads: 1
Cannot Cancel Request
Since a dasher has already picked up your laundry, your request is in progress and cannot be cancelled.
Cancel Request
🟩 Picked up Laundry
● Laundry in Washer
● Laundry in Dryer
● Dasher On Way for Drop Off
Cancel Request
Home

  - The **image on the left** is where the customer can see the page presented for the associated JobRequest when a user clicks on an item the table in **CustomerHomePage**. At the **top portion**, the customer can see the name and rating of the dasher assigned to the job, and the estimated cost based on the customer's inputted estimated number of loads. At the **bottom portion**, the user can see the current status of the laundry request, which is updated by the dasher throughout the completion of the job.
  - The **second image from the left** depicts the pop-up that is displayed to show the customer the final cost of the request, and allow the customer to leave a star rating and written review for the dasher. This pop-up is displayed after the dasher enters the updated cost information (actual cost required to complete the request) and confirms that the laundry was dropped off (job is complete). The input field on the top of the Receipt pop up

- The **third image from the left** depicts the pop-up that is displayed when the customer clicks the "cancel" button at the bottom of the screen. Clicking the "Yes, cancel it" button on this pop-up will delete this job request, and notify the dasher that the customer cancelled the request. Note that requests can only be cancelled before the laundry has been picked up by the dasher, otherwise the pop-up on the **image on the right** will be shown if the user clicks the cancel button after the laundry was picked up (button should say "okay")
-

- **DasherHomePage**
  - This page will allow a user logged in as a dasher to ask to be assigned a request, and to see a table with each of his or her in progress jobs.

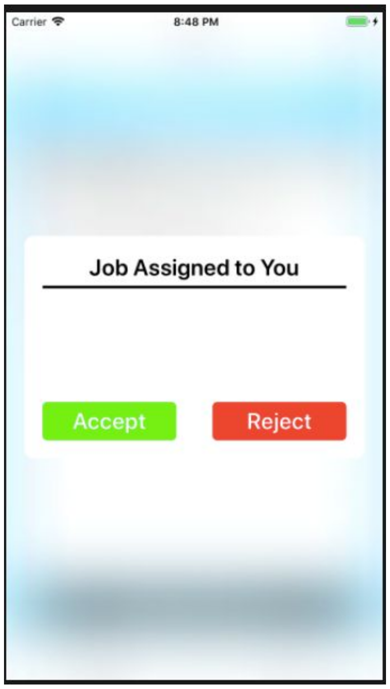


Laundry in Dryer
24CCF4E8-B6F0-465C-B794-11CDF5189637

Dasher on Way for Pickup
FC0F0D2E-D068-4FBB-8430-28122A4E8965

Dasher Accepted Request
F854D889-56E8-465B-A433-627D37ED529A

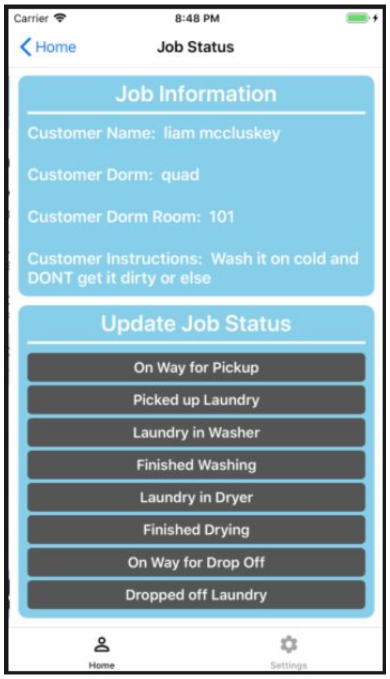
Dasher Accepted Request
71AEC980-ABF3-4819-A8CD-101BD88E32CF



  - The **image on the left** depicts the homepage a user sees when the user logs in successfully as a dasher. The button on the top right will present the **DasherJobsHistoryPage**. The main portion of this page is the table in the "In Progress Jobs" section of the page. This table contains an item for each in progress job the dasher has, and each item displays the current status of the JobRequest and the jobID of the JobRequest. Clicking on an item in this table will present the user with the **DasherJobStatusPage** for the JobRequest associated with the item in the table the user clicked.
  - The **image on the right** depicts the pop-up that is displayed after the user clicks the "Assign Me a New Job" button on the **image on the left,** and a job is assigned to the dasher (must be jobs available in the queue on database otherwise no job is

assigned). If a user clicks the “Accept” button on this pop-up, then the JobRequest is added to the table on the **image on the left**.

- **DasherJobStatusPage**
  - This page will allow the dasher to see the information about a JobRequest, and it will allow the dasher to update the current status of the JobRequest.



Job Information

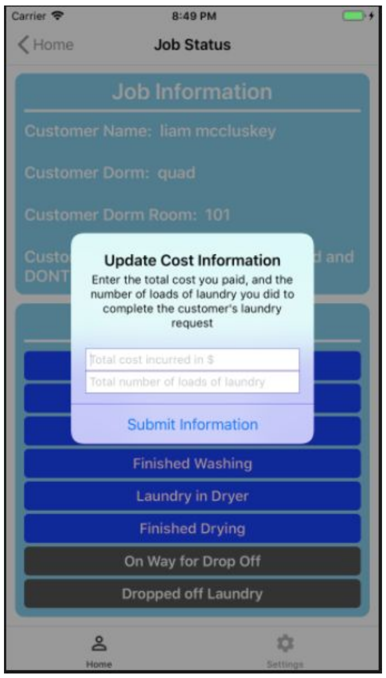
Customer Name: liam mccluskey

Customer Dorm: quad

Customer Dorm Room: 101

Customer Instructions: Wash it on cold and DONT get it dirty or else

Update Job Status



Job Information

Customer Name: liam mccluskey

Customer Dorm: quad

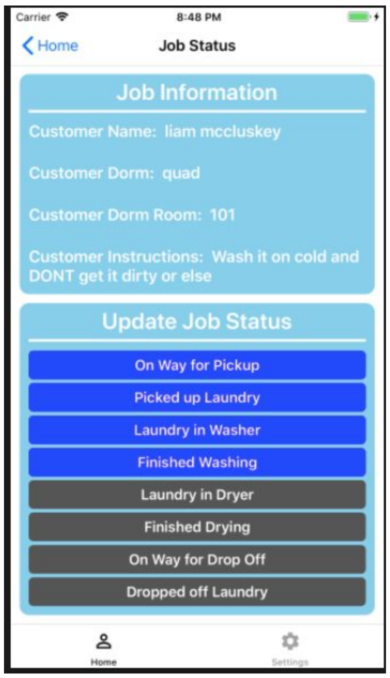
Customer Dorm Room: 101

Update Cost Information

Enter the total cost you paid, and the number of loads of laundry you did to complete the customer's laundry request

Total cost incurred in $

Total number of loads of laundry



Job Information

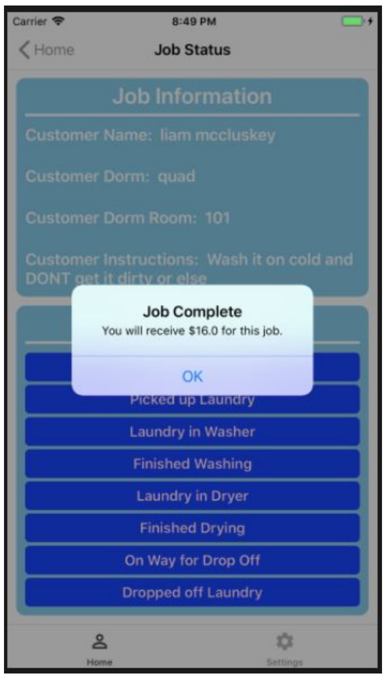
Customer Name: liam mccluskey

Customer Dorm: quad

Customer Dorm Room: 101

Customer Instructions: Wash it on cold and DONT get it dirty or else

Update Job Status



Job Information

Customer Name: liam mccluskey

Customer Dorm: quad

Customer Dorm Room: 101

Customer Instructions: Wash it on cold and DONT get it dirty or else

Job Complete

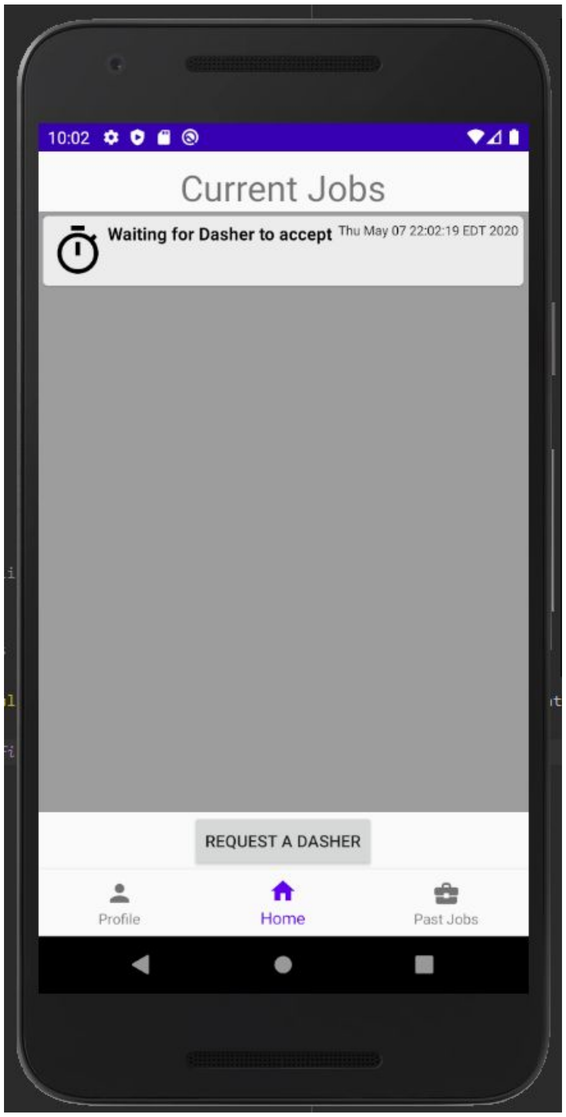
You will receive $16.0 for this job.

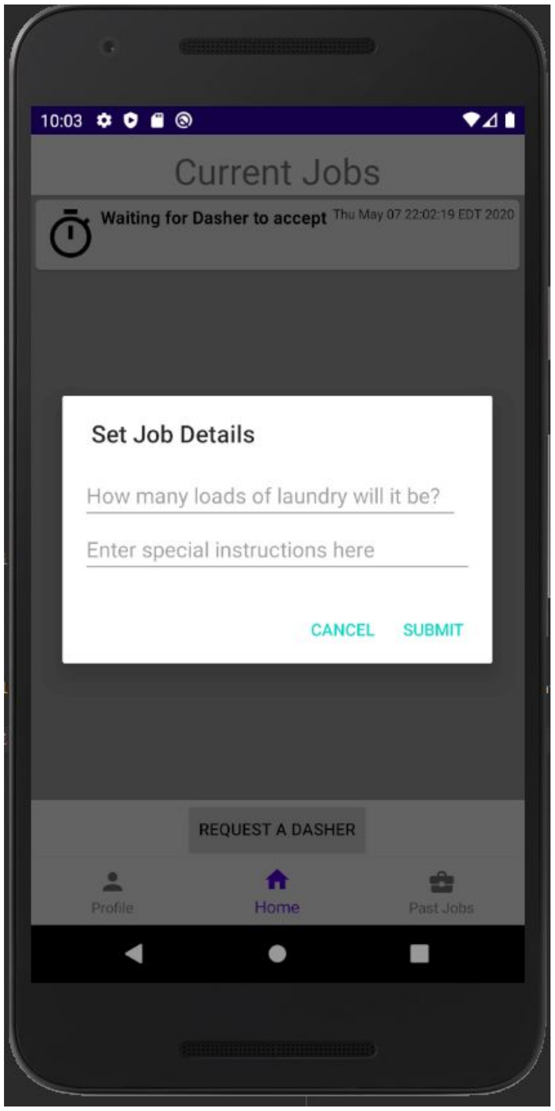
  - The **image on the left** depicts the page presented when a dasher clicks a JobRequest item in the table on **DasherHomePage**. The **top portion** of the page is where the dasher can see all the information about the request such as the customer’s name, dorm room, instructions, etc. The **bottom portion** of the page contains the buttons the dasher will press to update the status of the request at different stages. Buttons are initially grey, and turn blue when they are pressed which can be seen on the **second image from the right.**
  - The **second image from the left** depicts the pop-up that is displayed when the user clicks the “Finished Drying” button. This pop-up will allow the dasher to enter the updated information about how many loads the customer’s request actually required to process, and the total cost the dasher incurred in completing the request (cost of washing and drying).
  - The **image on the right** depicts the pop-up that is shown when the dasher clicks the “dropped off laundry” button. This pop-up displays the total amount of money the dasher will receive for this job (based on the updated cost information he or

she submitted). Clicking the "OK" button on this pop-up will remove the JobRequest from the **DasherHomePage** and close the **DasherJobStatusPage** for this job.

Android User Interface

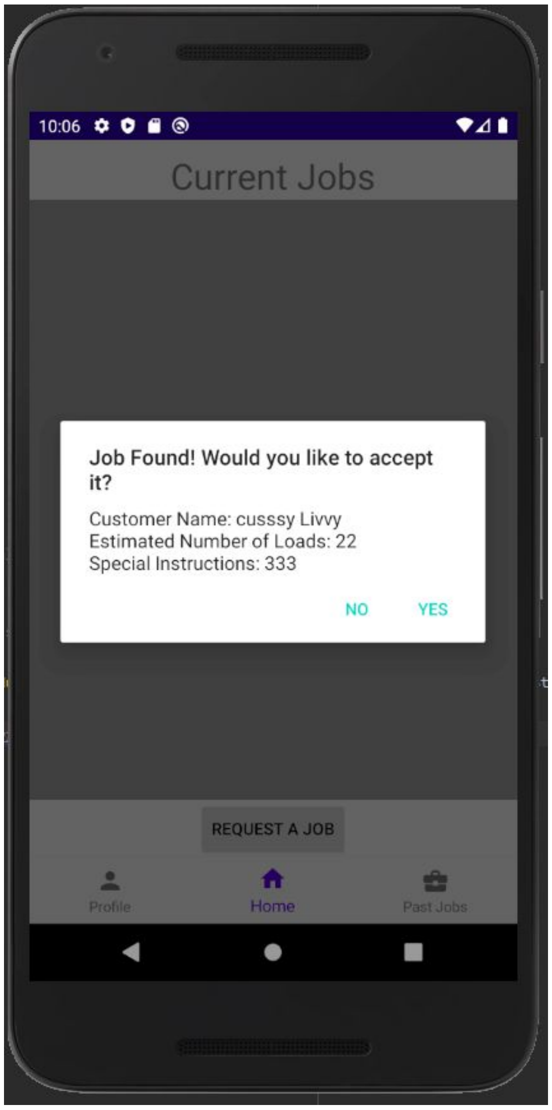
- CustomerHomePage



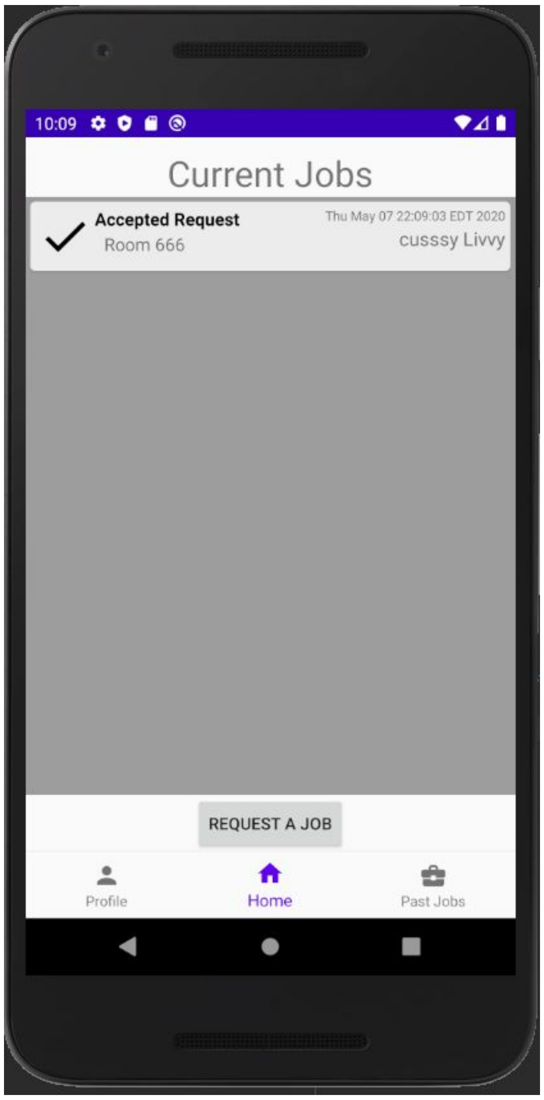

10:03
Current Jobs
Waiting for Dasher to accept
Thu May 07 22:02:19 EDT 2020
Set Job Details
How many loads of laundry will it be?
Enter special instructions here
CANCEL
SUBMIT
REQUEST A DASHER
Profile
Home
Past Jobs

-

- Dasher home



Customer Name: cusssy Livvy
Estimated Number of Loads: 22
Special Instructions: 333



Accepted Request
Room 666
Thu May 07 22:09:03 EDT 2020
cusssy Livvy

Design of Tests

Test Cases [4] :

Customer registration

1. Description: expected customer registration scenario - "sunny day" test
    a. User opens application and selects "Register an account" on sign in screen
    b. User presses "Register as a Customer".

    c. User then enters valid credentials to be sent to the database that meet the requirements of each respective text field
    d. User presses submit after completing each text field
2. Description: user attempts to register as a customer, however enters in data that is invalid or does not meet requirements - “rainy day” test
    a. User opens application and selects “Register an account” on sign in screen
    b. User presses “Register as a Customer”
    c. When filling in the text field for user credentials, the user will enter in some valid credentials, some invalid credentials, and leave some fields blank
    d. User presses submit, user is then redirected to the same registration page with invalid/empty text fields highlighted to show that they need to be filled in or meet a certain requirement
3. Description: user attempts to register as customer but enters text fields using different language following same steps as “sunny day” test, however enters in every text field in mandarin/emojis - (edge case)
    a. See above for steps at test case #1
    b. User presses submit, is then redirected to the same registration page with “invalid value” message above every text field that was entered in with mandarin/emojis

<u>Dasher Registration</u>

1. Description: expected Dasher registration scenario - “sunny day” test
    a. User opens application and selects “Register an account” on sign in screen
    b. User presses “Register as a Dasher”.
    c. User then enters valid credentials to be sent to the database that meet the requirements of each respective text field
    d. User presses submit after completing each text field
    e. User is redirected to laundry questionnaire page to verify laundry abilities: passes exam with a score of 80% or higher
2. Description: user attempts to register as a Dasher, however enters in data that is invalid or does not meet requirements - “rainy day” test
    a. User opens application and selects “Register an account” on sign in screen
    b. User presses “Register as a Customer”
    c. When filling in the text field for user credentials, the user will enter in some valid credentials, some invalid credentials, and leave some fields blank
    d. User presses submit, user is then redirected to the same registration page with invalid/empty text fields highlighted to show that they need to be filled in or meet a certain requirement

   e. User then submits with corrected text field values
   f. User is redirected to laundry questionnaire page to verify laundry abilities: passes exam with a score of 80% or higher
3. Description: expected Dasher registration scenario, however user failed questionnaire - "rainy day" test
   a. See above for steps from a-d at test case #1
   b. User is redirected to laundry questionnaire page to verify laundry abilities: fails exam with a score of less than 80%
   c. User prompted to wait 24 hours to retake test
      i. Attempts to retake test after an hour -> error message that x amount of hours remains before user can retake test
   d. After 24 hours, user retakes test and passes
4. Description: user attempts to register as customer but enters text fields using different language following same steps as "sunny day" test, however enters in every text field in mandarin/emojis - edge case test
   a. See above for steps at test case #1
   b. User presses submit, is then redirected to the same registration page with "invalid value" message above every text field that was entered in with mandarin/emojis

<u>User Log in</u>

1. Description: expected sign in credentials received and directed to home page - "sunny day" test
   a. User opens application and enters in correct and existing username and password
   b. User presses submit and directed to homepage
2. Description: user enters in incorrect username or password - "rainy day" test
   a. User opens application and enters in either an incorrect username or password
   b. User presses submit
   c. User is redirected to same page and informed that an incorrect username or password has been entered
   d. User retries entering in information within 5 tries limit and is successfully directed to home page
3. Description: user enters in non existing username - "rainy day" test
   a. User opens application and enters in a non existing username
   b. User presses submit
   c. User is informed "that username does not exist, you can register a new account here" (where here is hyperlinked to registration page)
   d. User registers new account (see registration test cases)

Customer Requests Job

1. Description: expected job request scenario - “sunny day” test
   a. Customer opens application and logs into account
   b. Customer selects “Request job” button
   c. Customer enters in text fields regarding information about job
   d. Customer is then matched with a Dasher and presented with the Dasher’s information to be accepted or denied -> user accepts
   e. Customer is then directed to Active Jobs page where can see updated information on the status of their job
   f. After job is completed, the customer confirms that job is completed on the active job page
   g. Customer then chooses to rate the dasher from 1-5 stars

2. Description: customer denies first Dasher presented - “rainy day” test
   a. Customer opens application and logs into account
   b. Customer selects “Request job” button
   c. Customer enters in text fields regarding information about job
   d. Customer is then matched with a Dasher and presented with the Dasher’s information to be accepted or denied -> customer denies
   e. Customer then presented with a new Dasher profile and selects accept
   f. See above test case #1 for remaining steps (steps e-g)
3. Description: customer cancels job after matched with dasher and before Dasher picks up - “rainy day” test
   a. See above test case #1 for steps a-e
   b. Customer cancels job prior to Dasher picking up laundry
   c. Job is effectively cancelled and customer redirected to home page
4. Description: customer cancels job after matched with dasher and after Dasher picks up - “rainy day” test
   a. See above test case #1 for steps a-e
   b. Customer cancels job after Dasher already picked up laundry
   c. Message screen appears informing the customer that the job can not be cancelled

Dasher Requests Job

1. Description: Dasher requests job and job is successfully executed - “sunny day” test

    a. Dasher opens application and logs into account
    b. Dasher selects “Request job” button
    c. Dasher presented with customer screen/job info and accepts
    d. Dasher directed to Active Jobs page where dasher can update status of job in realtime
    e. Dasher marks job completed after having entered in “on the way”, “wash”, “dry”, “on the way” in the job status
    f. Dasher rates customer from 1-5 stars
2. Description: Dasher requests job and denies first customer/job info presented - “rainy day” test
    a. Dasher opens application and logs into account
    b. Dasher selects “Request job” button
    c. Dasher presented with customer screen/job info and denies
    d. Dasher presented with new customer screen/job info and accepts
    e. See above test case #1 for steps d-f

Test coverage:

Through these outlined test cases, every activity and possible job path would be tested. We would have accessed, at least once, each activity/page. The activities that would be tested are as follows: SignInActivity, CustomerHomeActivity, DasherHomeActivity, RegisterActivity, CustomerRegisterActivity, DasherRegisterActivity, DasherProfileActivity, CustomerProfileActivity, CustomerReviewActivity, DasherReviewActivity, DasherJobCompletionActivity, CustomerJobCompletionActivity

Integration Testing[1]:

For this particular project, we are implementing a Top-Down approach to Integration Testing. This is because we are splitting different activities amongst the members of our group. The one problem with this is that we would have to wait for certain activities to be developed first, but that is traded with the advantage of testing these models before moving onto the lower level activities. For example, before having our Login Activity ready, we had to develop and test our Registration Activity, which is already split amongst the Customer and Dasher Role. Testing is done to make sure that a document is written into the correct collection in Firebase so the Login operation is possible. The modules that have the most dependencies on other activities and classes are done last, but they also tend to be the most problematic ones, since they can create unintended errors[4]. It is because of this that we plan to develop the higher level activities to the point where we can test them across multiple devices and see that they are functional, allowing us to implement the lower level activities with greater ease.

History of Work

<u>History</u>: At the beginning of the project, we had three to four meetings dedicated to picking a project idea that we wanted to work on. After deciding on our topic, the team would meet three to four times a week in order to flesh out the logistics and subtasks of our project. These weeks had the highest concentration of meetings.[3] Once subtasks had been assigned to different subteams, the team would meet once a week and the different sub teams would meet over Google Hangouts whenever they felt the need to. Now we only conduct meetings over Google Hangouts due to COVID- 19 and social distancing protocols.

<u>Current Status</u>: Core functionality of the iOS and Android apps has been completed. We have integrated payment for both iOS and Android using apple pay and Google Pay, respectively. The only thing left to be added before the final demo is the location tracker that will make sure Dashers are at or near their dorms when they accept a job.

<u>Future Work</u>: Since we have a working proof of concept down the next step would be implementing it into the real world so that it could be hopefully be used by millions of college students across the world. There would have to be an authentication system that made sure a user was in college when they registered. The system would also need to have information about all the dorms on each college campus. We will start out by adding all the dorms at Rutgers and performing extensive tests with multiple customers and dashers sending requests to the system.

<u>Key Accomplishments</u>:

- Created intuitive layout using Swift and Android Studio
- Efficiently integrated firebase to store and get all account and job information
- Created efficient algorithm that pairs customers with Dashers
- Found a relevant problem and proposed a solution for it

References

[1] Bradford, Laurence. “How to Make An Android App For Beginners - 5 Tips!” *Learn to Code With Me*, 7 Feb. 2020, learntocodewith.me/programming/android/beginner-app-development/.

[2] Samuyi. “How To Avoid Merge Conflicts.” *The DEV Community*, dev.to/samuyi/how-to-avoid-merge-conflicts-3j8d.

[3] Harvard Business Review Staff. “The Four Phases of Project Management.” *Harvard Business Review*, 3 Nov. 2016, hbr.org/2016/11/the-four-phases-of-project-management.

[4] Milano, Diego Torres. *Android Application Testing Guide: Build Intensively Tested and Bug Free Android Applications*. Packt Publishing, 2011.

[5] “Connect to Firebase : Android Developers.” *Android Developers*, developer.android.com/studio/write/firebase.

[6]harkiran78 “Top Programming Languages for Android App Development.” *GeeksforGeeks*, 19 May 2019, www.geeksforgeeks.org/top-programming-languages-for-android-app-development /.

[7] “Installation & Setup on Android | Firebase Realtime Database.” *Google*, Google, firebase.google.com/docs/database/android/start.

[8]Mayank. “How to Build an App like Uber: Uber for Drivers & Riders - EngineerBabu.” *EngineerBabu Blog & Success Stories*, EngineerBabu, 29 Jan. 2020, engineerbabu.com/blog/how-to-build-an-app-like-uber/.