Course Name: Software Engineering

*

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

Group #10

Drip ‘n Dash

Report 2

*

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

Group Members:

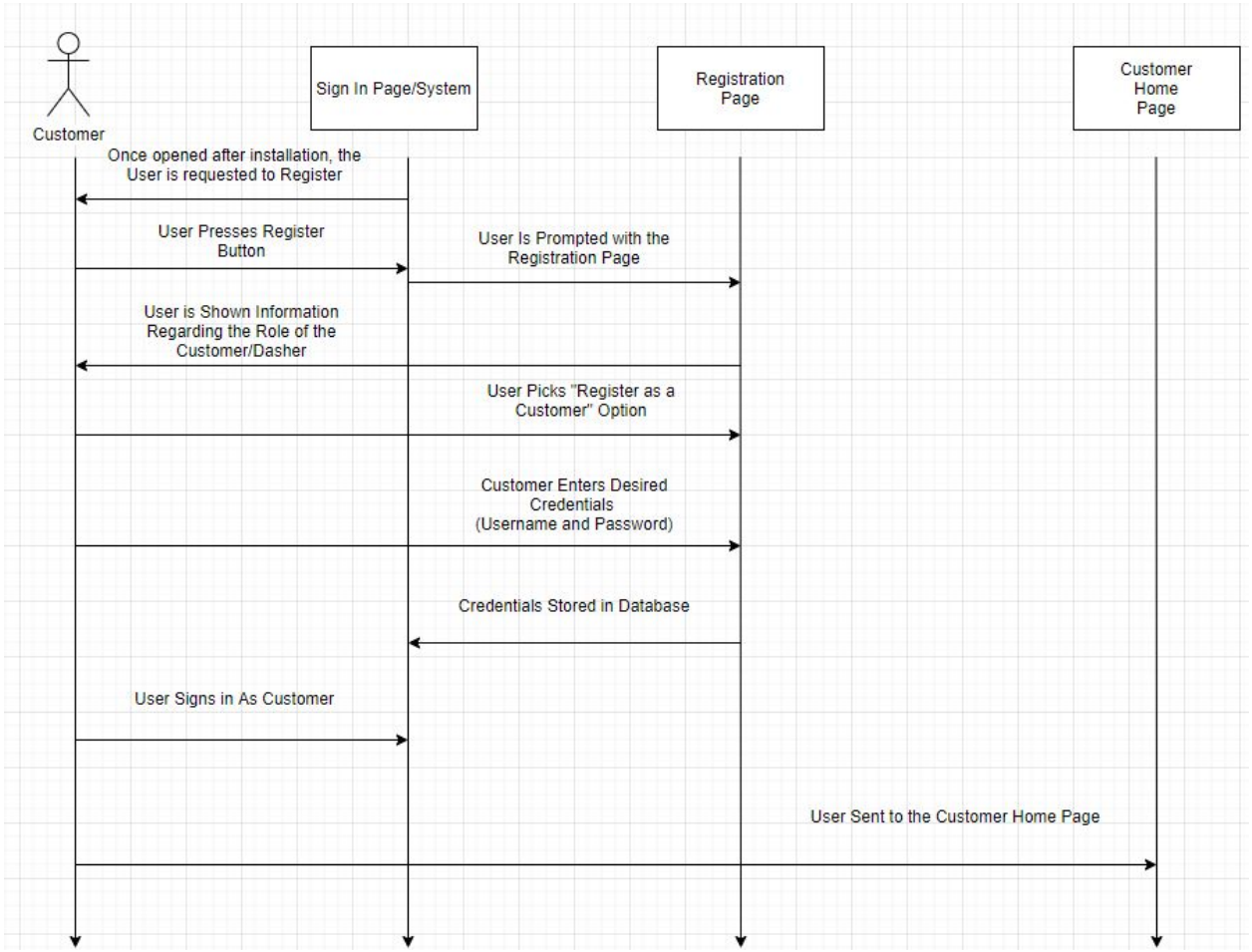
Elaina Heraty, Abdullah Bashir, Harshavardhana Dantuluri, HyunSik Kim, Karan Parab, Liam McCluskey, Nayaab Chogle, Nicolas Rubert, Roberto Cruz, Siddharth Manchiraju

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

1. Interaction Diagrams

UC: Customer Registration

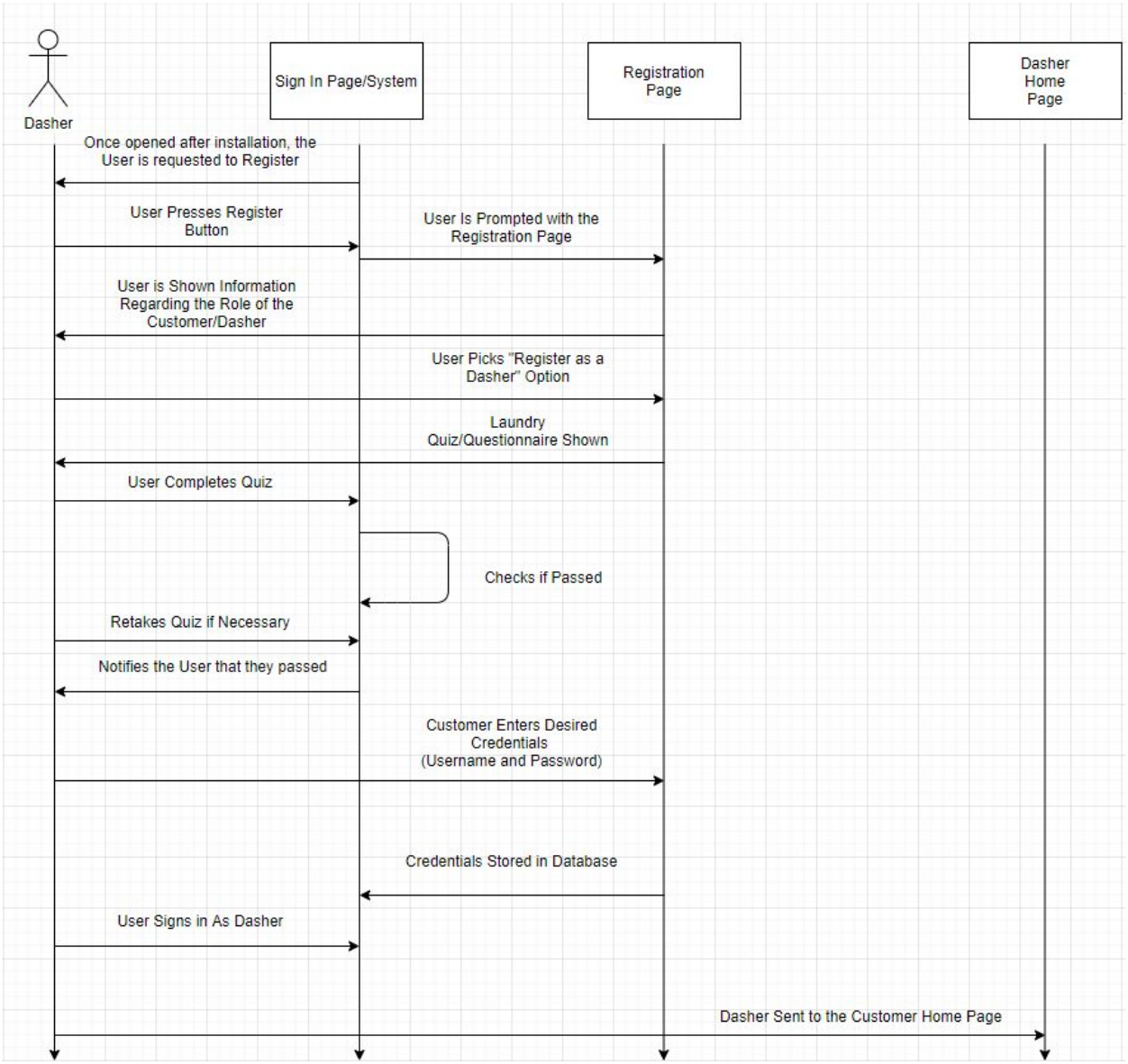


Design Principles:

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, the system will redirect 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 for the users.

UC: Dasher Registration

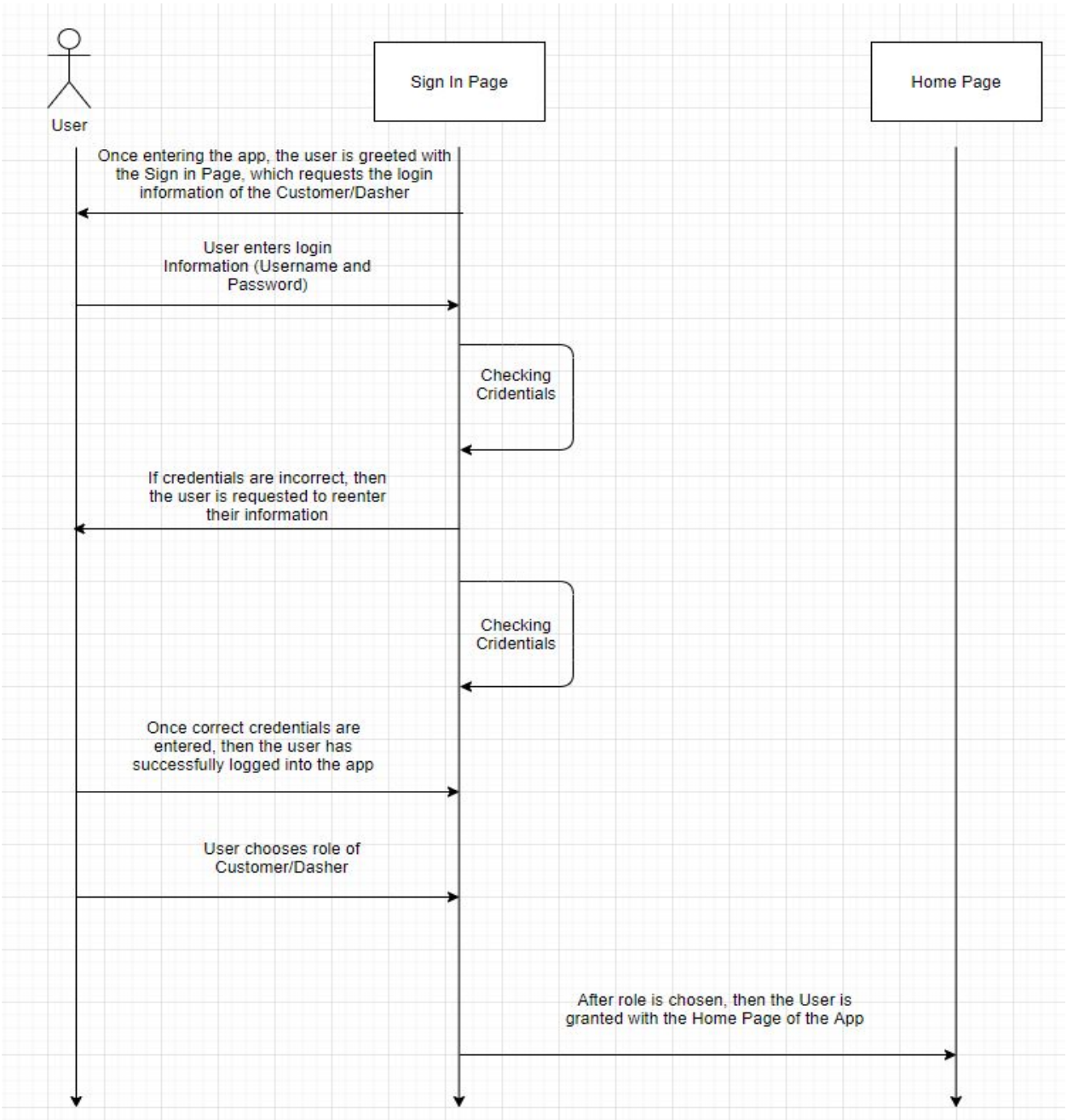


Design Principles:

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 ". When the dasher picks “Register as a Dasher”, the dasher needs to complete a quiz that evaluates the dasher if he or she qualifies as a

dasher. After that, the user's information will be needed to be registered as a Dasher. If the dasher completes the registration process, the system will redirect to the Sign In page and the customer 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.

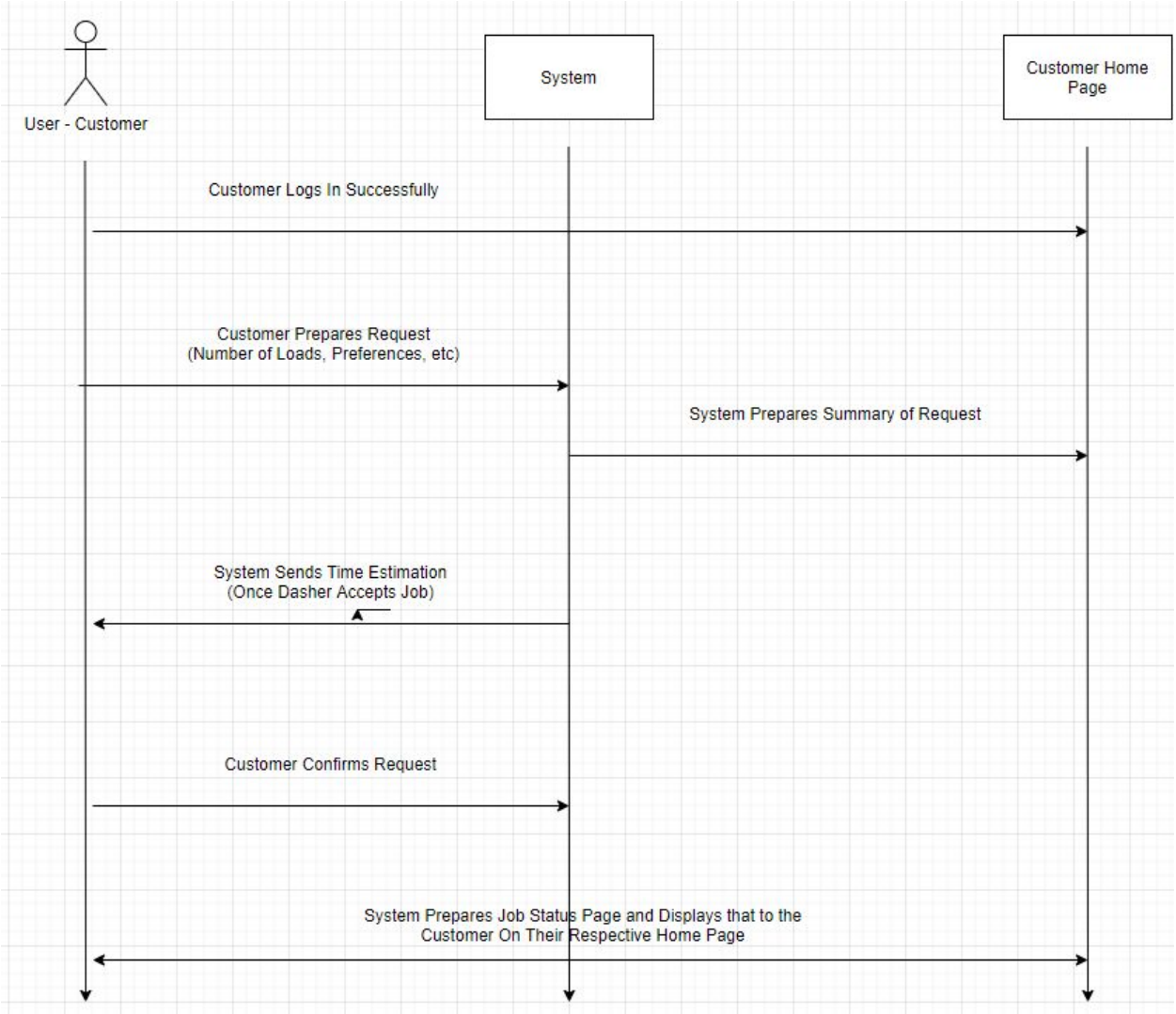
UC: CustomerLogin



Design Principles:

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 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.

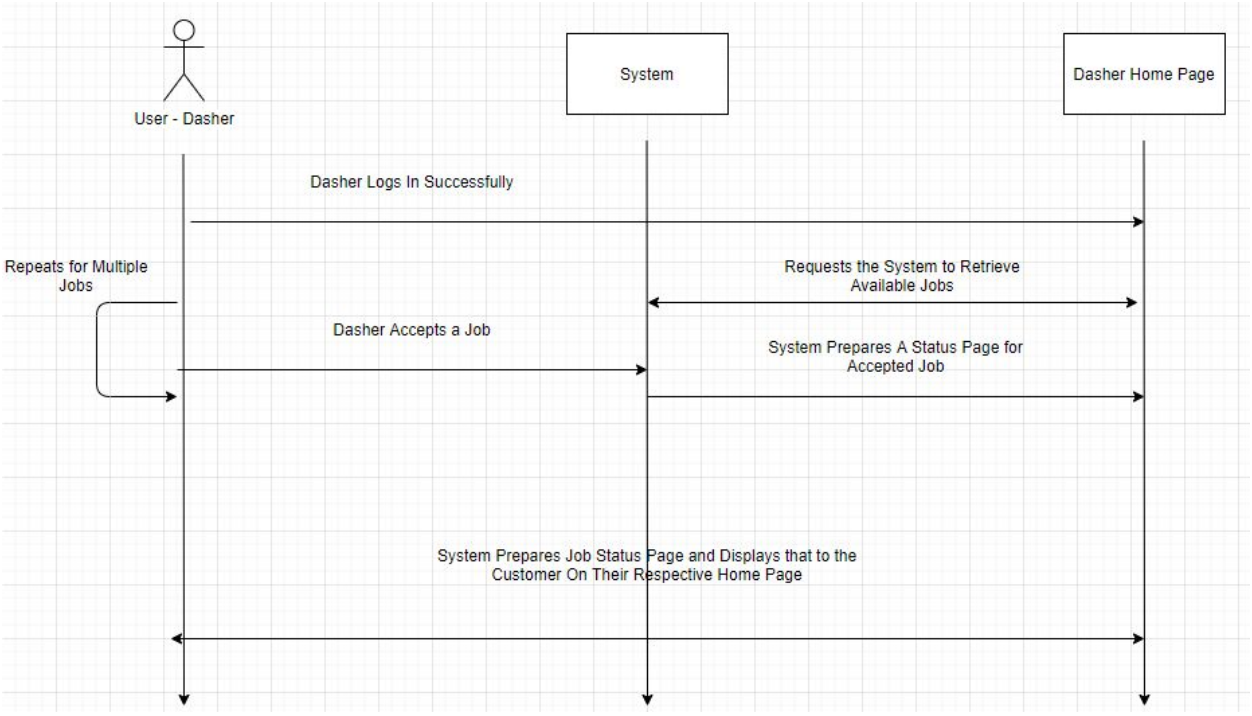
UC: CustomerSubmitRequest



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 (Request Time, 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.

UC: Dasher Accept Job

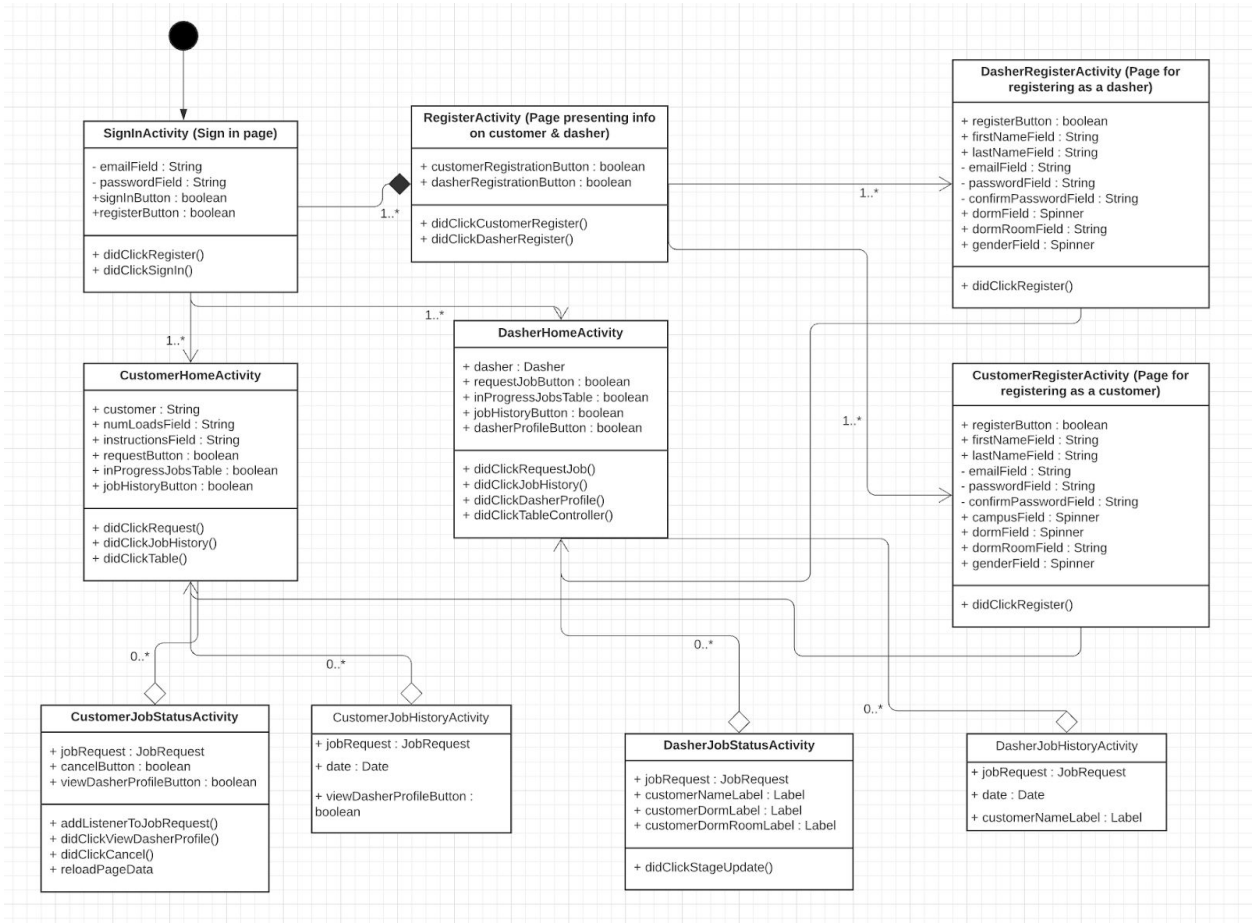


This figure shows the sequence diagram for Use Cases: Dasher Accept Job. If the dasher is successfully logged into the application. The app will show available jobs on the list and the request will be added to Dahser's in progress job list when the dasher accepts a request. This method is favored by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

2. Class Diagram and Interface Specification

---

a. Class Diagram



b. 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
}

- CustomerJobHistoryActivity (page for customer to see info on all his/her past requests)
DEPENDENCIES: Customer, CustomerFirestore
  - **Still developing this**
- DasherJobHistoryActivity (page for dasher to see info on all his/her past requests)
DEPENDENCIES: Dasher, DasherFirestore
  + **Still developing this**

- CustomerJobInformationActivity (page for customer to see all information on a past job request)
DEPENDENCIES: CompletedJob, Customer, CustomerFirestore
  + **Still developing this**

- DasherJobInformationActivity (page for dasher to see all information on a past job)
DEPENDENCIES: Dasher, DasherFirestore
  - **Still developing this**

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
}

c. 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

3. System Architecture and System Design

---

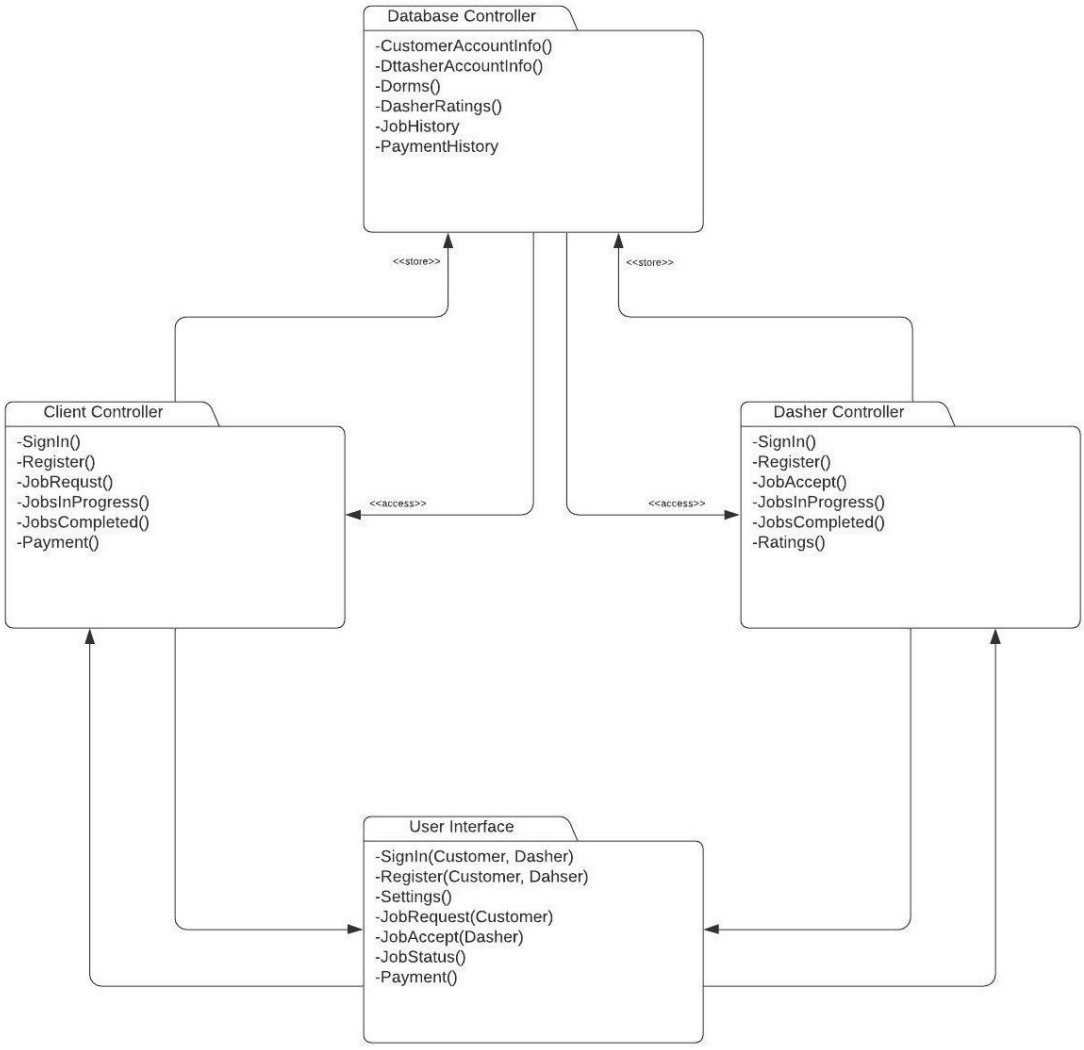
a. Architectural Styles

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.

b. 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.



c. 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.

d. 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)
- dorms (C)
  - dormName (D)
    - 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)

- 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)

e. 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.

f. 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.

g. 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 Android. 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.

4. Algorithms and Data Structure

---

a. Algorithms

i. 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

b. Data Structures

i. 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.

5. 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.

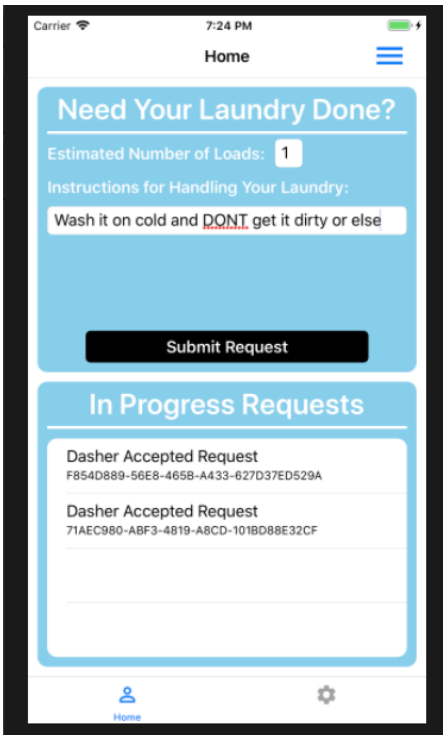
Unmodified (still undeveloped) Screen Mock-ups

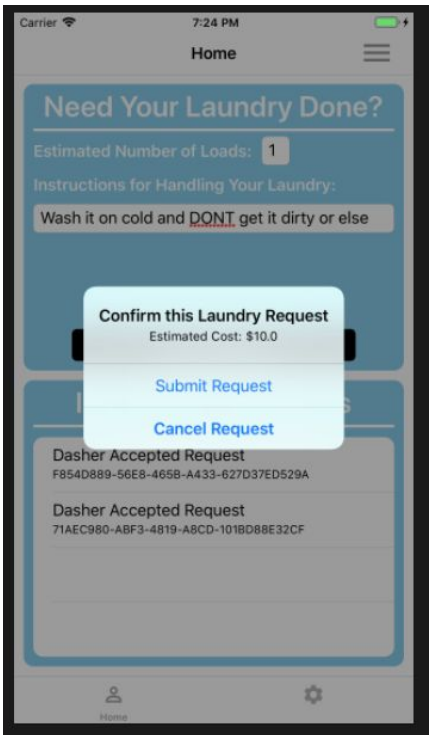
- CustomerJobsHistoryPage, DasherJobsHistoryPage, CustomerJobInformationPage, DasherJobInformationPage

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

Modified Screen Mock-ups[1]

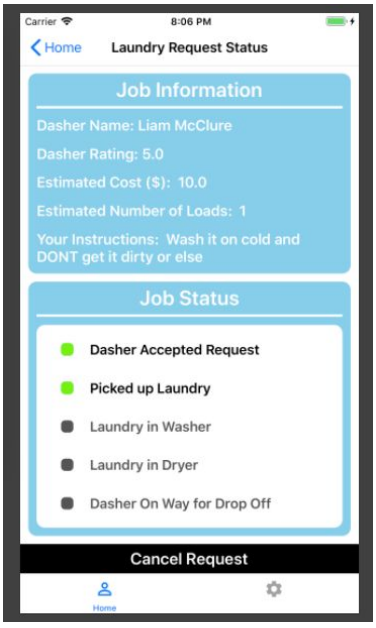
- **CustomerHomePage**
  - This page was modified to allow a user logged in as a customer to perform two core functions: submitting a request for a dasher to pick up laundry, and to see a table with all the customer's in progress jobs.

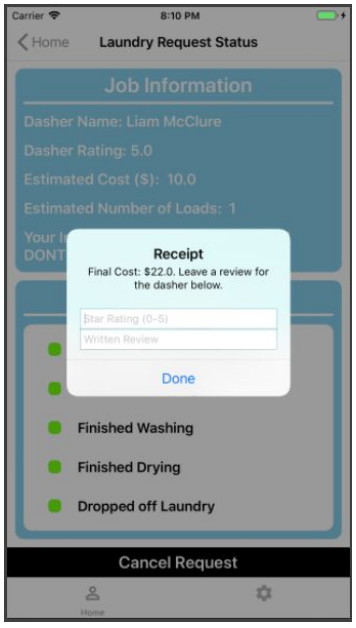


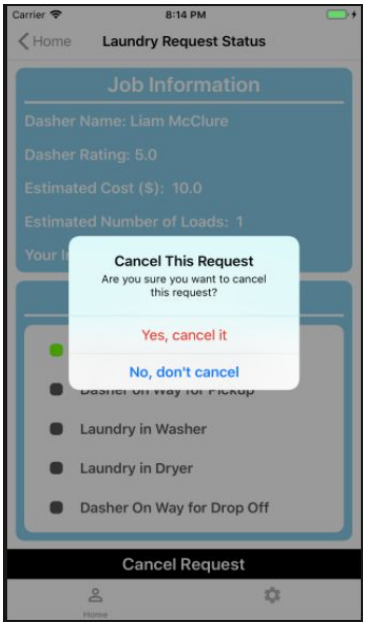


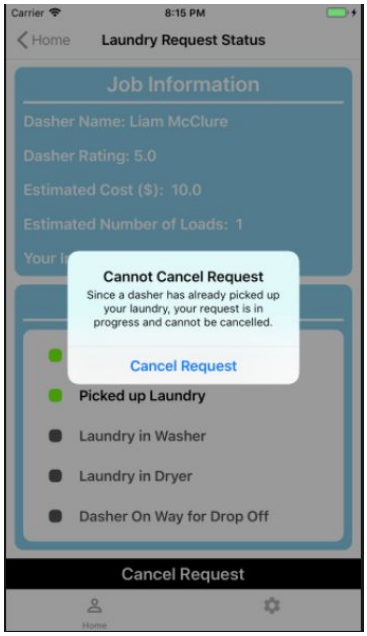
- The **image to the left** depicts the homepage a user sees after successfully logging in as a customer. The **top portion** is where the user can enter the estimated number of loads the laundry contains, and any instructions the dasher should follow when doing laundry. At the bottom of the “Need Your 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



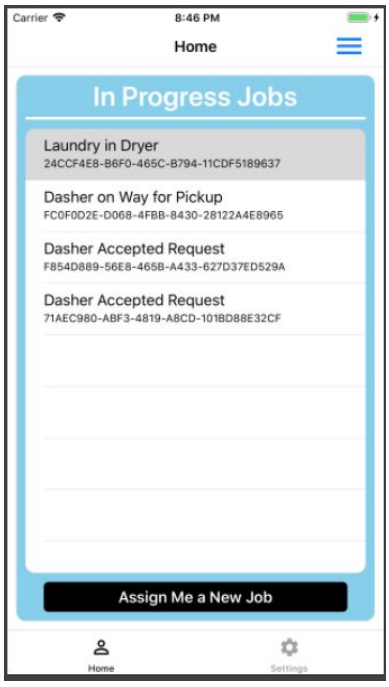


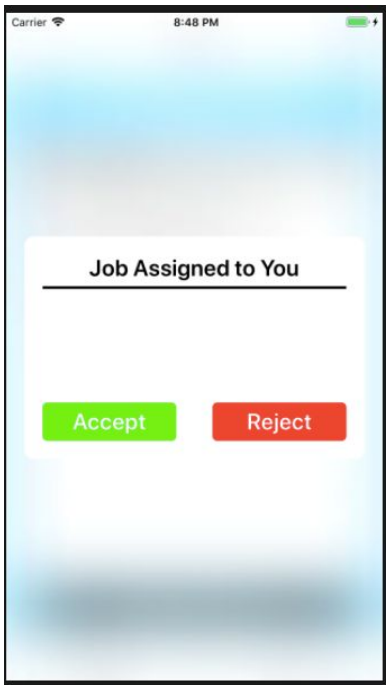




-
- The **image on the left** is where the customer can see the page presented for the associated JobRequest when a user clicks on an item on the table in **CustomerHomePage**. At the **top portion**, the customer can see the name and rating of the dasher assigned to the job, and the estimated cost based 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")
-

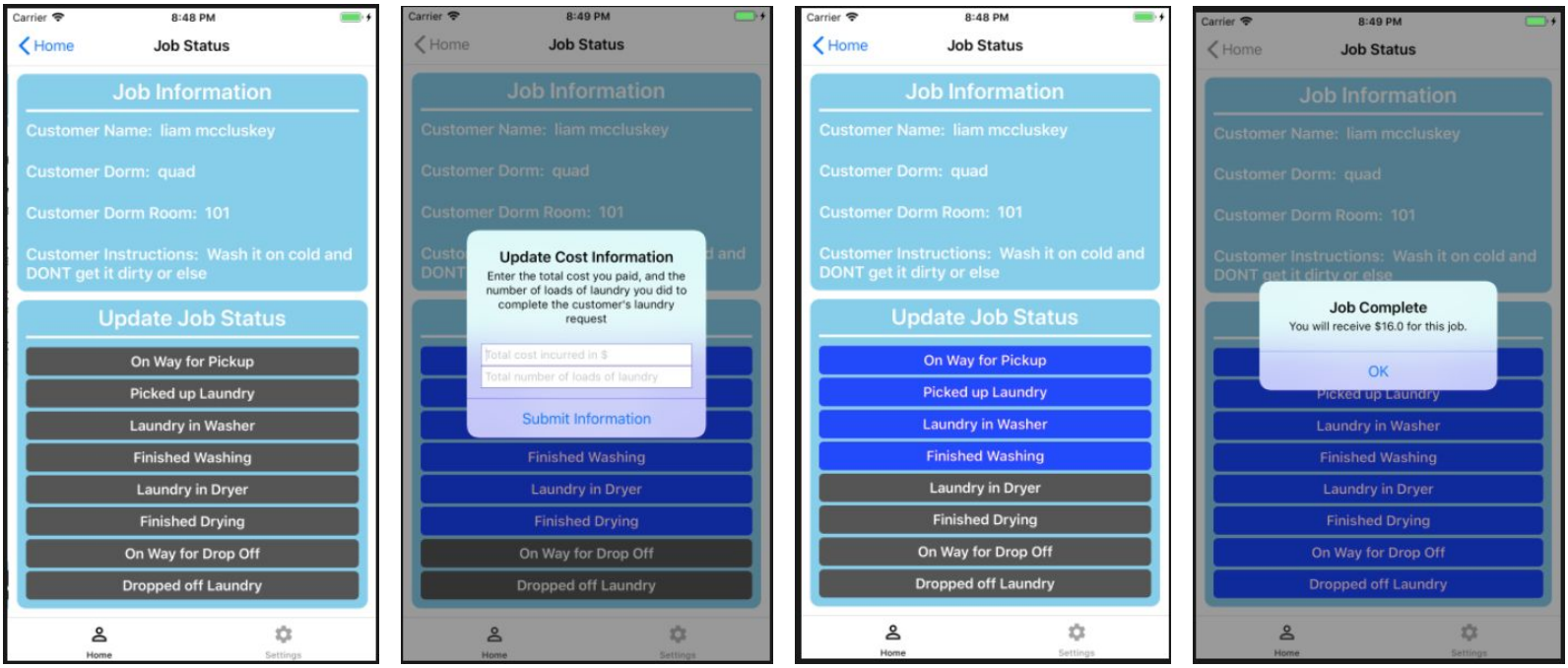
- **DasherHomePage**
  - This page will allow a user logged in as a dasher to ask to be assigned a request, and to see a table with each of his or her in progress jobs.





- 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.



- 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.

6. 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

Dasher Registration

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:

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.

7. Project Management and Plan of Work

---

a. Merging the Contributions from Individual Team Members

We were able to avoid any sort of merge issues by simply making sure only one person was working on a file at a time[2]. By working this way, we were able to always push to the master branch, avoiding any confusion that comes with having to manage multiple branches. Merge issues would only arise if multiple members had made changes inside the doc. If a member had trouble pulling, they would simply discard any changes they had made to the file causing merging issues. Since each file is worked on by one member at a time, the changes made would likely be accidental and negligible, meaning that they would be okay to discard[2].

All the pseudo code containing methods, classes, objects, and variables is contained in a google doc accessible to the team. By following the conventions in this doc, the team is able to keep all code consistent across the project. All the object classes were created first to ensure that other classes containing methods that required the objects worked. Appearance is kept uniform by adhering to Android Studio's default appearance settings and colors. Once, all of the functionality is complete, members of the team interested in design will begin to discuss what kind of colors they want and the like.

b. Project Coordination and Progress Report

Completed Use Cases: User Create/Login, Job Creation and Job Acception, Job Updates, and Job Completion.

Completed Functions: Registration, Sign In, customers requesting a Dasher, Dashers requesting for a job to be assigned to them. This is most of the core functionality that has been completed.

Functions currently being completed: Review system, Past Jobs, ability to edit Profile information.

Weekly meetings are held to ensure that all team members are up to speed on how the code works[3]. Each member presents their current progress and explains how their code works. We make sure that every member has the most up to date version of the code from the repository and that the code is running properly on their system.

c. Plan of Work

Task name	Assignee	Due date	Priority
Reports			
Full Report #2 1		Today	Medium
Full Report #3		May 3	Low
Reflective essay		May 3	Low
Electronic Project Archive		May 7	Low
Implementation			
Optimize Database Structure 3	lm liam mccl	Mar 2	High
Log in	HK Hyun Sik Ki...	Mar 5	High
Registration 1	NR Nicolas Ru...	Mar 18	High
Sending Jobs	RC Roberto Cr...	Mar 19	High
Receiving Jobs	NC Nayaab Ch...	Mar 16	High
Status of Job	SM Sid Manchi...	Mar 19	Medium
Settings	EH Elaina Hera...	Mar 18	Low
Menu	HD Harsha Da...	Tomorrow	Medium
Payment (Venmo) Integration	AB Abdullah B...	Mar 19	Medium
Ratings/Reviews	AB Abdullah B...	Tomorrow	Low
Design(color scheme)	KP Karan Parab	Wednesday	Low
Testing			
Initial Demo		Apr 1	Medium
Final Demo		Apr 27	Low

d. Breakdown of Responsibilities

SignInActivity - Abdullah and Liam
RegisterActivity - Robert
CustomerRegisterActivity - Abdullah
DasherRegisterActivity - Abdullah
CustomerHomeActivity - Liam and Harsha
DasherHomeActivity - Abdullah and Liam
CustomerJobStatusActivity - Nayaab and Harsha
DasherJobStatusActivity - Elaina and Harsha
DasherProfileActivity - Karan
CustomerFirestore - Nayaab

DasherFirestore - Abdullah
JobRequestFirestore - Nicolas and Liam
CustomerDasherProfileActivity - Elaina and Liam
CustomerReviewActivity - Nayaab and Harsha
DasherJobCompletionActivity - Siddharth and Karan
CustomerJobHistoryActivity - Liam and Hyunsik
DasherJobHistoryActivity - Robert and Nicolas
CustomerJobInformationActivity - Elaina and Liam
DasherJobInformationActivity - Robert

Liam and Nicolas will handle coordinating the integration of the modules and classes. Each member of the team will be responsible for integration testing. We chose Liam and Nicolas because they understand the most about the different working parts of the whole application. They understand how the database we are using, Firebase, works as well. Each student will test the modules that they were responsible for, and the team will meet together to test how their parts work together[3]. We will use Git to compile the parts.

Each member tests their own code to make sure it works. Then they test to see if their code works with the rest of the project since everyone is working out of the same repository. After individual testing, each member pushes their changes on the master branch of Github Desktop. In this way, every member of the team is responsible for testing.

8. 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.