

DIGITAL ACCESS TO SCHOLARSHIP AT HARVARD
DASH.HARVARD.EDU



HARVARD LIBRARY
Office for Scholarly Communication

# Shared Shopping List

**The Harvard community has made this article openly available. Please share how this access benefits you. Your story matters**

| | |
|---|---|
| Citation | RASULI, DAVID. 2019. Shared Shopping List. Master's thesis, Harvard Extension School. |
| Citable link | https://nrs.harvard.edu/URN-3:HUL.INSTREPOS:37365091 |
| Terms of Use | This article was downloaded from Harvard University's DASH repository, and is made available under the terms and conditions applicable to Other Posted Material, as set forth at http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA |

David N. Rasuli

A Thesis in the Field of Software Engineering

for the Degree of Master of Liberal Arts in Extension Studies

Harvard University

08 / 2018

Copyright 2018 David N. Rasuli

## Abstract

Shared Shopping List is a mobile application which aims to help collaborative shopping using mobile phones. This thesis project presents a solution to assist group of users such as families, friends and organized group to conduct shopping for multiple users. While such solutions already exist, this project will attempt to demonstrate usage of cutting-edge technologies such as React-Native, and new cloud services such as NoSQL DynamoDB, and Lambda, along with Serverless software architecture approach to support a scalable, and collaborative multi user application.

# Dedication

To my Parents, Evelyn & Saad Rasuli. Great believers and promoters of education. Dear Mother, I've never dreamt that a month before submitting this document, I will sit in the hospital with you and explain the term "Serverless Computing" at 2AM while waiting for the doctor on duty. Thanks to you and to my father, I carry the importance of lifelong learning, and keep expanding my knowledge.

Acknowledgments

First and foremost, I would like to thank my thesis director, Mr. Eric James Gieseke, for enlightening in me the passion for clear software design, cloud computing, and providing professional and personal guidance towards every step of this thesis project, for being available to provide all the assistance that I needed to complete this thesis, and for believing in me and in this project, even when I had doubts about my abilities and outcomes - It was a privilege, and I am honored. Thank you.

This project could not have been completed without the help of my wife, Ester Nahum-Rasuli, who made herself available to endless stay up late nights while I was working on this thesis project, and yet kept me focus on my mission to finish it, believed in me even when I thought of withdrawing, you are my strength and my inspiration.

For my parents, Evelyn & Saad Rasuli :
When I didn't want to study, you emphasized the importance of it.
When I struggled to study, you made sure that I will have the help I need.
When I wanted to study, you supported endlessly.
I've been blessed to be your son.

To my brother, Eyal Rasuli - thank you for continuingly believing in my innovative thinking.

To Ms. Natalie Azulay, Mr. Efraim Cohen and Rabbi Eli Friedman. Some reasons should stay behind curtains, but gratitude should be shown to all audience. Thank you.

Table of Contents


Abstract 3

Dedication 4

Acknowledgments 5

List of Figures 10

Introduction 12

Background 13

Prior Work 15

"AnyList" 15

"Google Keep" 15

"Buy me a pie" 16

"Trello" 16

Requirements 17

Functional Requirements 17

Collaborative : 17

Mobile: 17

Ease of use: 18

Non Functional Requirements 18

Performance : 18

Security: 18

Availability: 19

System Overview 20

High Level Architecture 20

Use Case Diagrams 22

Shopping List Use Cases: 22

Domain Model - High Level 24

Design and Technology choices: 25
Serverless architecture 25
AWS Lambda Functions: 25
Scalability: 25
Deployment: 26
Modularity and Flexibility: 26
User Interface 26
React-Native: 26
Ionic: 27
Cloud Computing 28
The Lambda Service: 28
The IAM - Identity & Access Management 29
The API Gateway Service: 29
The DynamoDB Service: 29
Programming Languages 30
Image Management: 30
Implementation 31
User Interface - UI Panels 31
Design via Tabs and Stack-Menu: 31
Settings Page : 36
User's Shopping Lists : 37
Shopping List Details : 38
Item Details : 39
Back End - Services and Web Methods 40
Items Management 41
Append Item To Shopping List : 41
Update Item In List : 43
Participant Management 44

Get Participant By Id: 44
Edit Participant: 44
Get List Participant By Id: 45
Shopping List Management 47
Change Shopping List Administrator: 47
Create Shopping List: 48
Duplicate Existing List: 48
Edit Shopping List: 50
Finish Shopping List: 51
Get List By List Id: 51
Application Utilities 52
Data Access: 52
Request Handler : 54
Design Choices 56
React-Native: 56
DynamoDb: 56
Application usefulness 57
Learnings 58
Summary and Conclusions 60
Features 60
Postponed : 60
Unplanned : 61
Data Model 61
Commercialize 62
Accomplishment 62
Plans for the future 63
Appendix. 65
Payload examples 65

Append Item To Shopping List 65
Update Item In List 65
Get Participant By Id 66
Edit Participant 66
Get List Participant By Id 66
Change Shopping List Administrator 67
Create Shopping List 67
Duplicate Existing List 67
Edit Shopping List 68
Finish Shopping List 68
Get List By List Id 68
References **69**

List of Figures

Figure 1: Application high level component diagram.

Figure 2: Application high level use case diagram.

Figure 3: Participant use case diagram.

Figure 4: Domain Model Diagram.

Figure 5: UI Panels - Navigation directed graph.

Figure 6: UI Panels - Tab Navigator.

Figure 7: UI Panels - Stack Navigator & User List.

Figure 8: UI Panels - Stack Navigator & Shopping List Details.

Figure 9: UI Panels - Stack Navigator & Item Details.

Figure 10: UI Panels - Settings Page.

Figure 11: UI Panels - User List.

Figure 12: UI Panels - Shopping List Details.

Figure 13: UI Panels - Item Details.

Figure 14: Component Diagram - Serverless

Figure 15: Sequence Diagram - Append Item To Shopping List.

Figure 16: Sequence Diagram - Get List By Participant ID.

Figure 17: Sequence Diagram - Duplicate Existing List.

Figure 18:  Component Diagram - Data Access.

Figure 19: Component Diagram - Lambda.

# Introduction

It is without question that mobile applications now support almost any information related task. This project will create a mobile multi user application for the management of shopping lists to help improve the shopping experience for consumers. This project will emphasize best practices for system design, by applying serverless architecture, the scalability of NoSql technologies, and exploring a relatively new way of developing mobile applications using React-Native. In addition, the design of the mobile user interface is optimized for easy and collaborative use by consumers. This project will include features from other multi user shopping systems that exist today, such as images and comments for products, and will show that serverless architecture can be applied to multi-user mobile applications. The resulting application will be referred to as the "Shared Shopping List".

## Background

Handwritten shopping lists are traditionally a very popular means for managing shopping and serve as a reminder while shopping for any product, from daily groceries to electronics and hardware materials. As mobile technologies have advanced, software applications such as "Don't Forget The Milk", "AnyList" and "Keep" have been created to provide an easy and digital way to manage shopping lists with the option to track the progress of the shopping.

The problem with these applications is that most of them are for self-use only. Imagine a scenario while shopping, where your partner discovered items missing from the original list, or that you, as a shopper, found out that some of the products are out of stock. A phone call could solve this, but if one's attention is already focused on the mobile's shopping list, it would be great if one could add and remove products and notify it to all the subscribers of the shopping list, especially if the buyer will conduct the shopping for an arranged group of shoppers.

Shared Shopping List should allow both the shopper and the subscribers to add, remove, modify list items and notify each other about changes to the list. The shopper could let the subscribers know what items are already in the cart and if a product is missing, suggest replacements. In addition, the system should support tracking, creating, loading, saving and editing shopping lists.

The goal of this project will be to create a cloud hosted solution which will manage products, users and shopping lists, and provide an API that serves the data to an end client mobile application for multiple users.

The project will focus on three design principles: cloud services, mobile integration, and simple UX/UI.

## Prior Work

This section describes prior work in the digital shopping list space.

### "AnyList"

AnyList is a mobile application that handles list item editing and reminder via scheduling systems. Besides lists management and scheduling, it provides item priority and tagging support, along with synchronization to Microsoft outlook. This application provides all the necessary lists and items management features that one wants, but is limited to self-use only. The Shared Shopping List application, as the name suggests, will be available for both self and multiple users.

### "Google Keep"

Google Keep is a shared list management application. Google Keep is a generic list management system, not limited to shopping lists A user can create a list and edit list of items such as text messages, images, and audio. One or more participant can share the same list; all of the participants can edit the list simultaneously. Shared Shopping List application will focus on shopping list, as opposed to a generic list application.

"Buy me a pie"

The Buy me a Pie application can manage items, quantity, and measure units of any shopping lists. It also provides a suggestion mechanism and simple UI, and also allows participants to update the lists. It does not allow image sharing of potential products and notes about items aren't available. The Shared Shopping List will support additional details about items, including images and comments which will increase user engagement during the shopping experience with the rest of the participants.

"Trello"

Trello is a general missions and tasks management application, which works both on web and mobile, could be shared via multiple participants and allow prioritizing of mission. Similar to Google Keep, Trello is a general purpose task management tool which has a UI that isn't shopping based, but rather tasks based.

## Requirements

This section reviews the requirements for the Shared Shopping List application.

### Functional Requirements

Collaborative:

The application should allow multiple users within a group to- view, update, and comment on the list, and one administrator to manage the list and conduct the shopping. All of the users in a specific shopping group can add items to the list, update notes, and images. In addition, an administrator can close a list (check-out), remove, invite and change administrator, and mark items as available or not. The idea of the application is to let the shopper be the administrator of the shopping list. A shopping list can be cloned to a new list, and have all the original items added to the cloned list. The shopping list and items can be edited, including the name and administrative rights of the list, and the item's name, quantity and measurement unit.

Mobile:

Due to the mobility required for shopping, the application should be a mobile application, so that the shopper can use their mobile phone to manage the shopping list. Mobile phones provide a convenient platform for the  the Shared Shopping List application due to their ubiquity, accessibility and network connectivity. .

Ease of use:

The goal is to make shopping easy, instead of writing countless text messages when shopping for a group of people, it would be better to have a simple and easy to use and navigate mobile application, that will be composed of a minimal number of screens, with two possible configurations - participant and administrator.

## Non Functional Requirements

This section will review the non functional requirements including performance, security, and availability.

Performance:

To allow a smooth user experience, each web request and push notification should talk less than one second to respond.

Security:

At the mobile application level, there is no passing of monetary transaction, and personal information that being saved at the backend won't consist any payment data of any form. Should a business model of this application will include user based data, then names, emails and other identifying data will not be disclosed. - Protecting of the information of the user. Users of this application can only see list which they are created or invited to,

and cannot be exposed to lists, items or participants which they aren't affiliated with. When a new list is duplicated, all the comments of the copied items from the previous list aren't being copied to prevent disclosing comments of previous participants which not necessarily take part of the newly duplicated list. Communication security for RESTful client to server such as token per sessions should take place once this application goes live, but was not introduced on this solution.

Availability:

This application assumes that users will constantly use this application, then on production release it will always have to be fully available, even through deployment. To allow this, the application assumes that the API gateway will always be available and so does the DynamoDB and Lambda functions.

## System Overview

This section of the document will provide a high level overview of the system components of the Shared Shopping List application.

## High Level Architecture

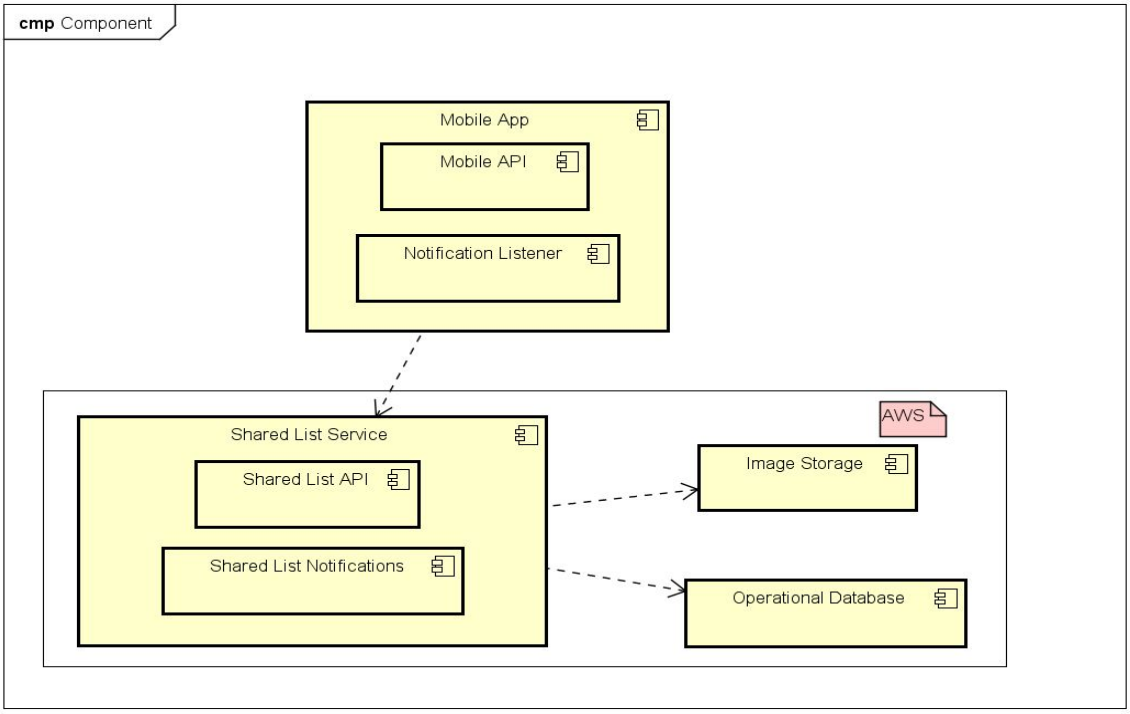



Figure 1 : Shared Shopping List application high level component diagram

The above diagram describes the high level components of the Shared Shopping List application and their interdependencies. The components work together to support the requirements. Each of the components are described below.

Mobile App: Represents a single user mobile list service application. Resides on user's mobile phone. Provides a user interface that interacts with the Shared List Service.

Mobile API: The client side component of the application which calls the back-end services.

Notification Listener: Receives notifications from the backend push services, and updates the UI state accordingly.

Shared List Service: Backend of the application, deployed over AWS to server users both via pull and push mechanisms.

Shared List API: Expose secure HTTPS REST interface for use by the Mobile API.

Shared List Notifications: Pushes messages via Firebase FCM channels for registered Notification Listeners.

Operational Database: The system's database, which holds operative data about users, lists , items in lists and items.

Image storage: A service which supports uploads and persistence of images and their URLs.

Use Case Diagrams

Shopping List Use Cases:

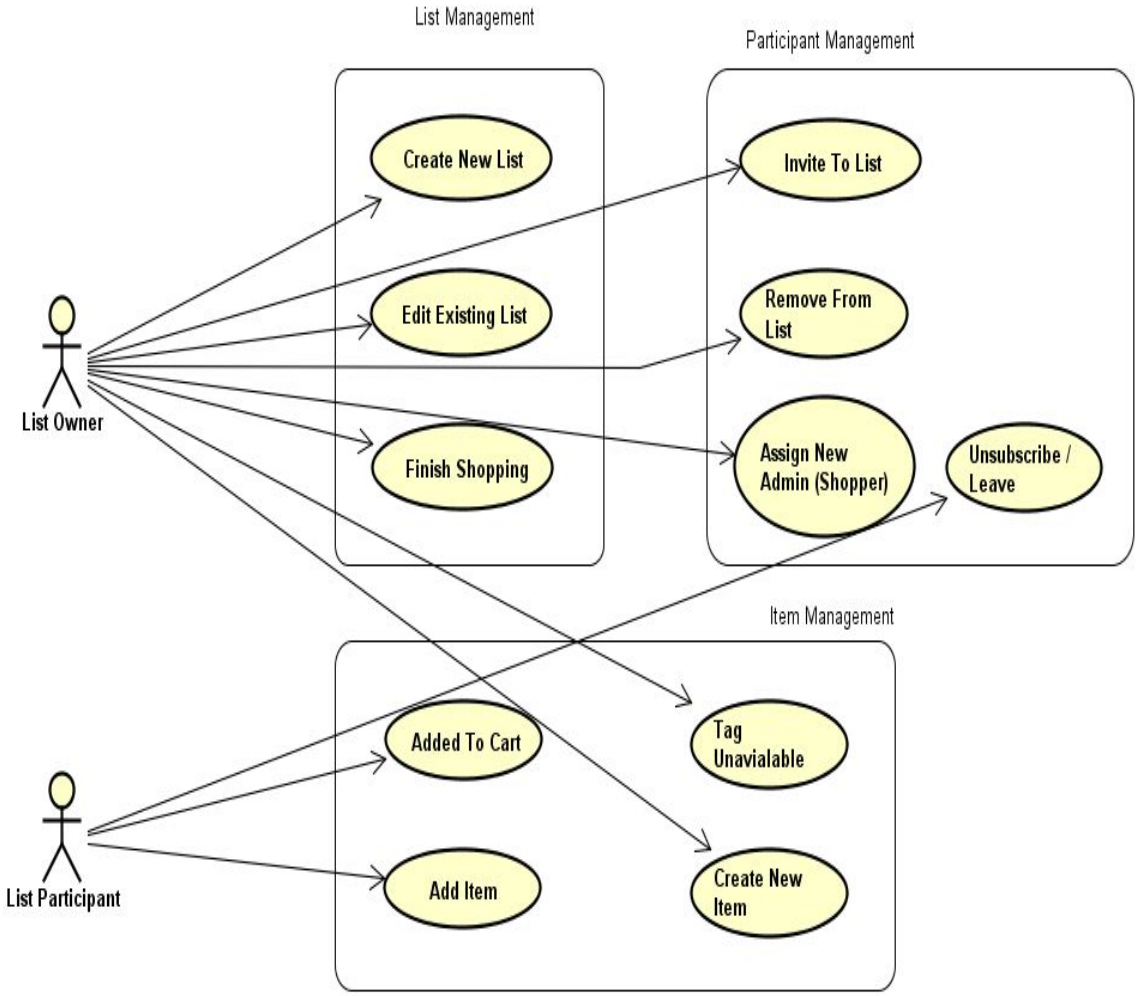



Figure 2 : Application high level use case diagram.

The above use case diagram divides the use cases into three categories with two roles: List Management - list alteration cases, Participant Management - any action that involves user within a specific list, and Item Management, which includes item management within a specific list.

Personal:

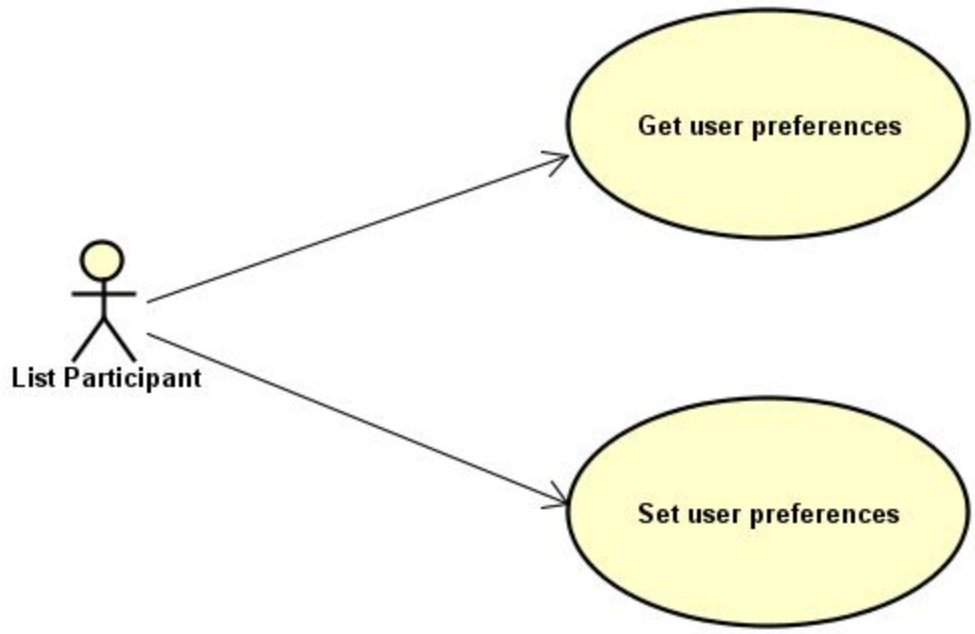



Figure 3 : Participant use case diagram.

A List Participant can get its personal information data, which includes a portrait image, email, and nick-name, and edit those accordingly.

Domain Model - High Level

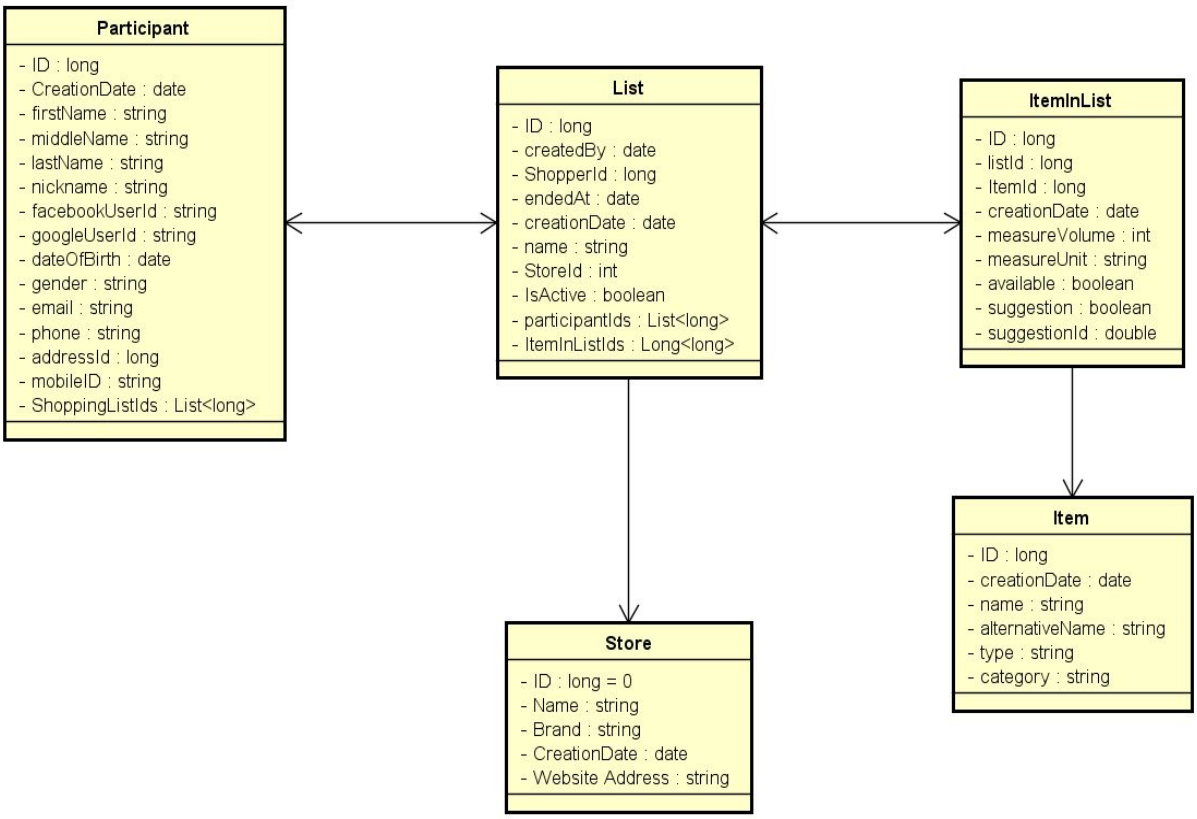



Figure 4 : Domain Model Diagram.

List holds details of any shopping list that has ever been created and managed at the system. It has Many-to-Many relationship with Participants which can be either Administrators of a list or simply participants. Item represents a shopping list item (e.g. milk) ItemInList connects a List with an Item, and is aware of which list it resides in. Store holds the data about the store of which the shopping is conducted at.

## Design and Technology choices:

### Serverless architecture

#### AWS Lambda Functions:

One of the goals of this project is to explore the application of serverless architecture to create a fully functional, scalable, flexible and easy to maintain application. AWS Lambda was chosen to be the serverless platform to deploy and run serverless web methods along with AWS API Gateway to configure and expose these Lambda functions through a REST interface. Each Lambda function represents a backend web method that supports one of the project use cases.

#### Scalability:

One of the major ideas behind serverless architecture is to allow unlimited scalability without explicit management of machines and load balancers. When a Lambda function is deployed and used, it will scale automatically up or down to support the load, leaving the management of the resources to the AWS infrastructure. This is known as elastic scalability and saves resources and simplifies service management.

Deployment:

Deployment of the backend Java code will use maven deployment packages. The package will be uploaded manually via AWS console or Docker. After a successful upload, the Lambda services are ready and will be configured as a REST Web Service via the API Gateway.

Modularity and Flexibility:

Since there is no need to manage servers, and each Lambda service stands on its own, it is easier to add new services and modify existing functions without impacting other functions.

## User Interface

React-Native:

A relatively new development tool for building mobile application, React-Native was created and released by Facebook in 2015. React-Native supports creating both Android and iOS applications using React web development methods, and extends it to mobile devices, including state management. The core feature that distinguishes React-Native from other development platforms such as Ionic and Cordova, is the use of native

components of both iOS and Android, and unlike the latter, which use a native browser which serves as a mobile compatible web application.

The use of native components allows better rendering, especially when graphic and client compute actions take place at the device, making it closer to an application that was written in a native code of iOS and Android platforms.

Once a React-Native application is compiled and packaged, it can be deployed as either an iOS application, using the XCode compiler, or an Android application, using the Android Studio Emulator.

For this project, I've chosen to implement an Android version.

There is an abundant amount of information about the React-Native platform from Facebook Development guides, open source libraries, and React-Native contributors such as WIX.com.

Ionic:

Ionic is another mobile application platform that allows creation of iOS and Android applications. The original thesis proposal included development of the UI components using Ionic., The author has decided to switch to React-Native, due to its relevance, efficiency, and pioneering approach in the mobile development field.

## Cloud Computing

All of the cloud computing components of this project are hosted within the Amazon Web Services (AWS) public cloud, using the following AWS services:

The Lambda Service:

AWS Lambda is the serverless solution which allows management of serverless functions. AWS Lambda allows functions to be written in a variety of programming languages, including Java which was used in this project. Functions can be deployed and tested via AWS Lambda Console and designated tools, allowing testing and monitoring of each function.

AWS Lambda has a “Cold” and “Warm” scenarios - “Cold” refers to a start of concurrent execution of a Lambda Functions - when a function is triggered for the first time or after some period of dormancy, the cloud provider initializes a container to run the function. Whereas “Warm” refers to a scenario in which the function has already initialized and running. As long as the functions are “Warm”, then the web-methods at this project takes less than a second to run.

When a function is “Cold”, it will take several seconds to run, in production environment, the servers will should have a serverless plugin such as “serverless-plugin-warmup”, to keep the function warm and allow smooth user experience.

The IAM - Identity & Access Management

The AWS Identity & Access Management (IAM) Service is designed to secure access to the cloud services. All the of the Lambda web requests deployed via AWS Gateway API, and accessing the DynamoDB entities will require usage of IAM role of which the only role permission is allowed solely to the author of this project.

The API Gateway Service:

AWS API Gateway allows deployment of REST based web-services endpoints, and is used to expose Lambda functions. The AWS API Gateway supports REST API deployment, environment configuration, monitoring and security at any scale.

The DynamoDB Service:

AWS DynamoDB is a fully managed, automatically scaled, non relational database which supports both key value and document storage. The AWS console supports configuration and management of the DynamoDB Service

The original thesis proposal included RDS as the database solution, but since scalability is a concern with an SQL based database, a NoSql option is a more scalable solution for concurrent read/write access to the database.

## Programming Languages

The back-end components were developed using Java. I've selected Java to allow faster development - as this is the most used and familiar language which the author masters. Additional Java libraries from AWS SDKs for DynamoDb and Lambda, and parsing utilities are being used at the core code of this project. The languages Python and NodeJS don't require compiling and Maven packages are faster to develop and deploy, however Java was chosen due to better familiarity.

The front end mobile application was written with JavaScript in conjunction with the React-Native framework, a React based platform which use states to modify data within the front end application. React-Native supports HTML like components that rendered to implement actual native components features.

## Image Management:

Images can be taken either by camera or by browsing the mobile device for existing images. The image link is saved to the DynamoDb when updating an Item. Future implementation will contain a back-end service such as S3 to allow the images visibility across all the participants of the list.

# Implementation

## User Interface - UI Panels

Design via Tabs and Stack-Menu:

Ease of navigation is a key concept for creation of a successful web and mobile application. Therefore, I've chose to split the user interface into two main categories - Shopping Lists & Settings, using Tab Navigator, and sub categories within those main categories using Stack Navigator.

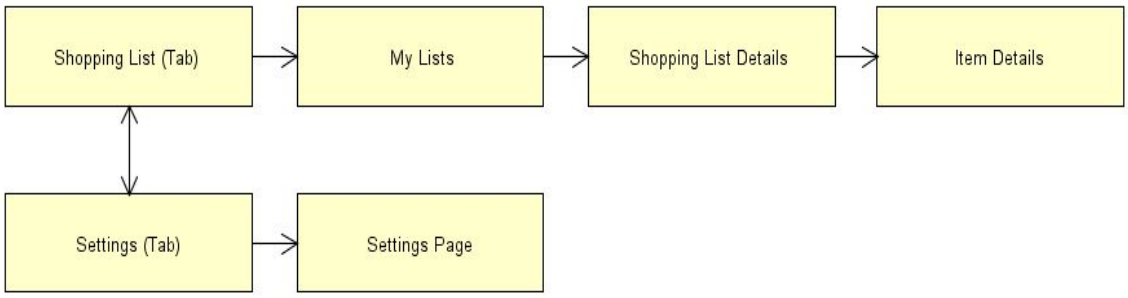



Figure 5 : UI Panels - Navigation directed graph.

The mobile UI has two tabs, Shopping List and Settings. Each of these tabs holds a stack navigator, allowing navigation back and forth from each screen. Besides providing ease of navigation, tabs and stacks navigator helps preserving the client stored data within the same context. I.e.: Pressing "Back" on Item Detail will return to the Shopping List which this item belongs to.

Tab Navigator - choses between Shopping Lists view and Setting View.

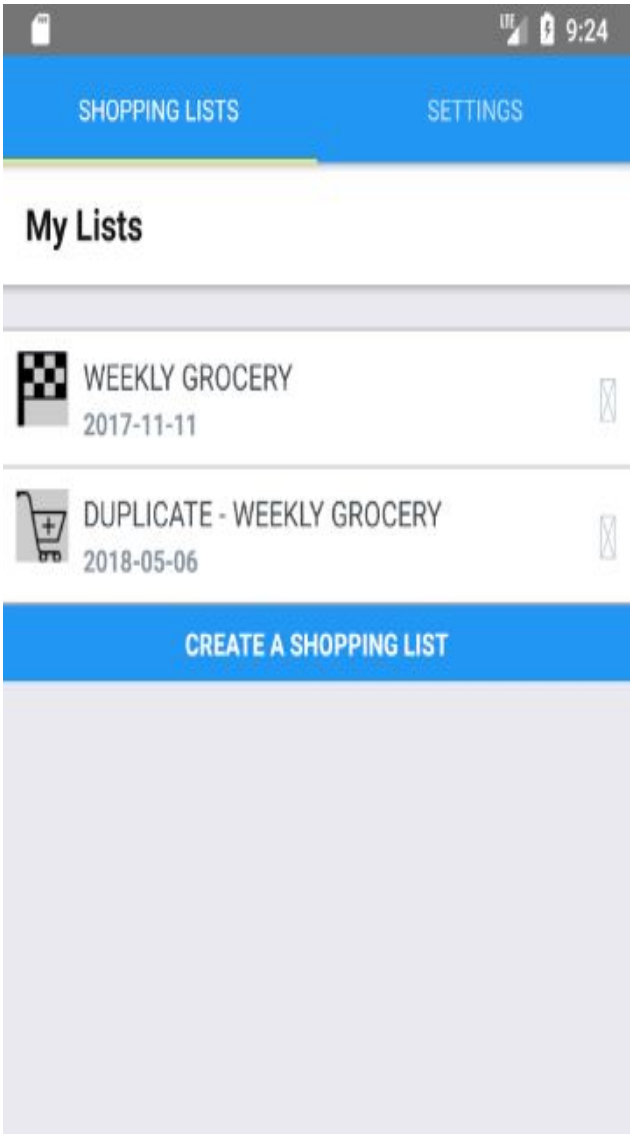
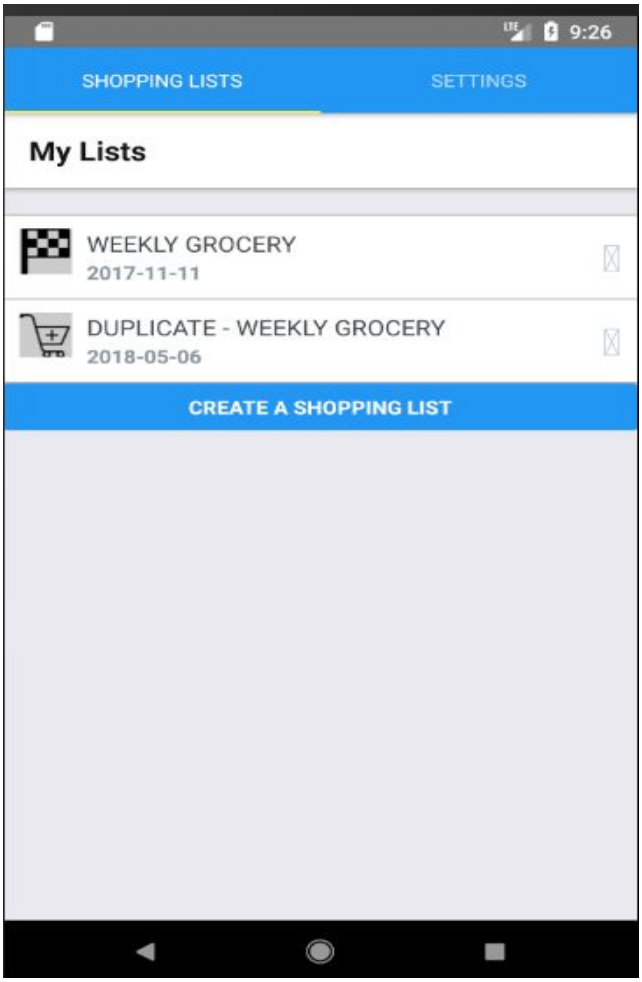



Figure 6 : UI Panels - Tab Navigator for accessing Shopping List and Settings

Stack Navigator - Allows navigating between Shopping List. Use the “Back” button to navigate from a Shopping List back to the list of Shopping Lists.

“Back” button :




At Shopping List tab, lets click on of the lists - “Weekly Grocery”.

Figure 7 : UI Panels - Stack Navigator & User List.

Now we're on "Weekly grocery" Shopping List, lets dive deeper and choose one of the items. If we would use the Tab Navigator and choose settings and then back Shopping list, we will arrive to this specific screen, and not to the initial "My Lists" screen.

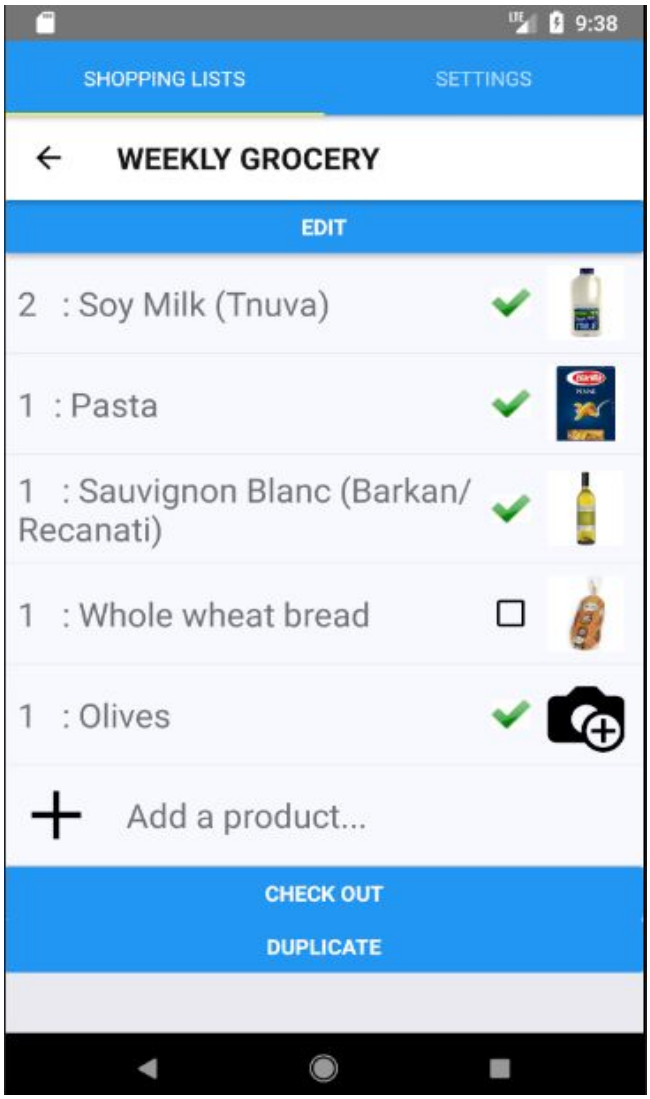



Figure 8 : UI Panels - Stack Navigator & Shopping List Details.

Once choosing an item, we could navigate back to the Shopping List using the arrow just by “Soy Milk”, below the Tab Navigator.

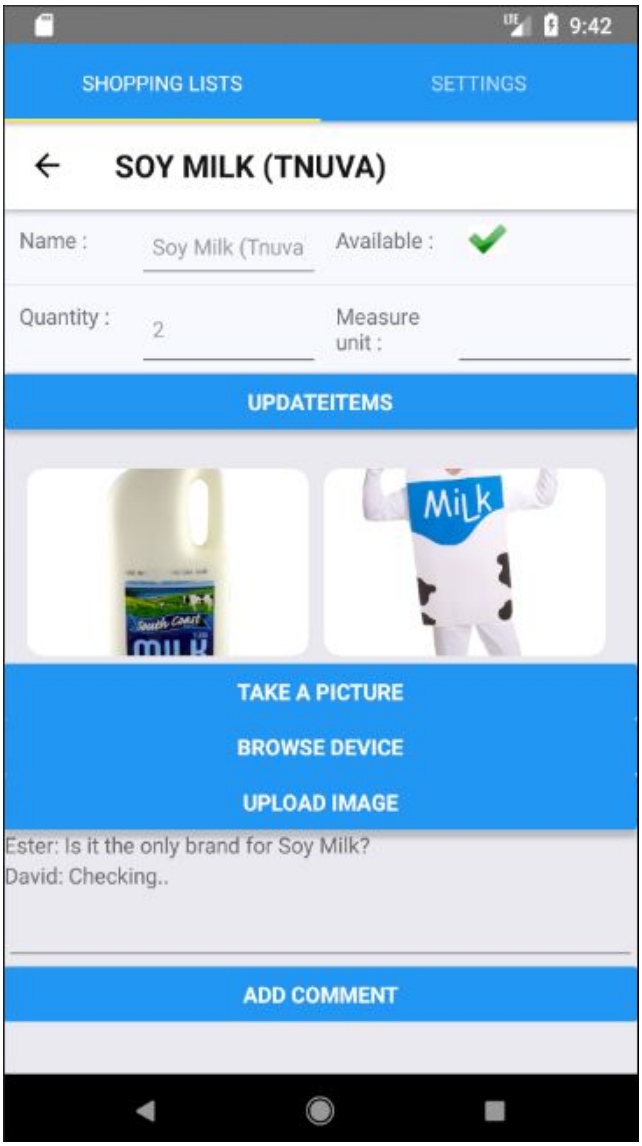
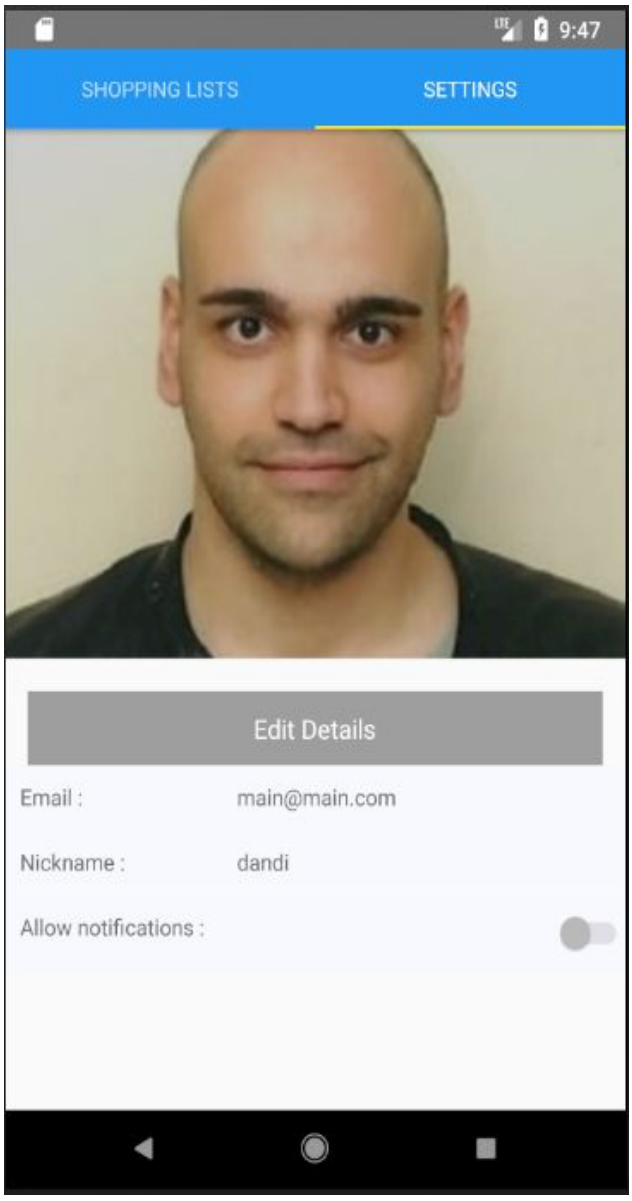



Figure 9 : UI Panels - Stack Navigator & Item Details.

Settings Page :

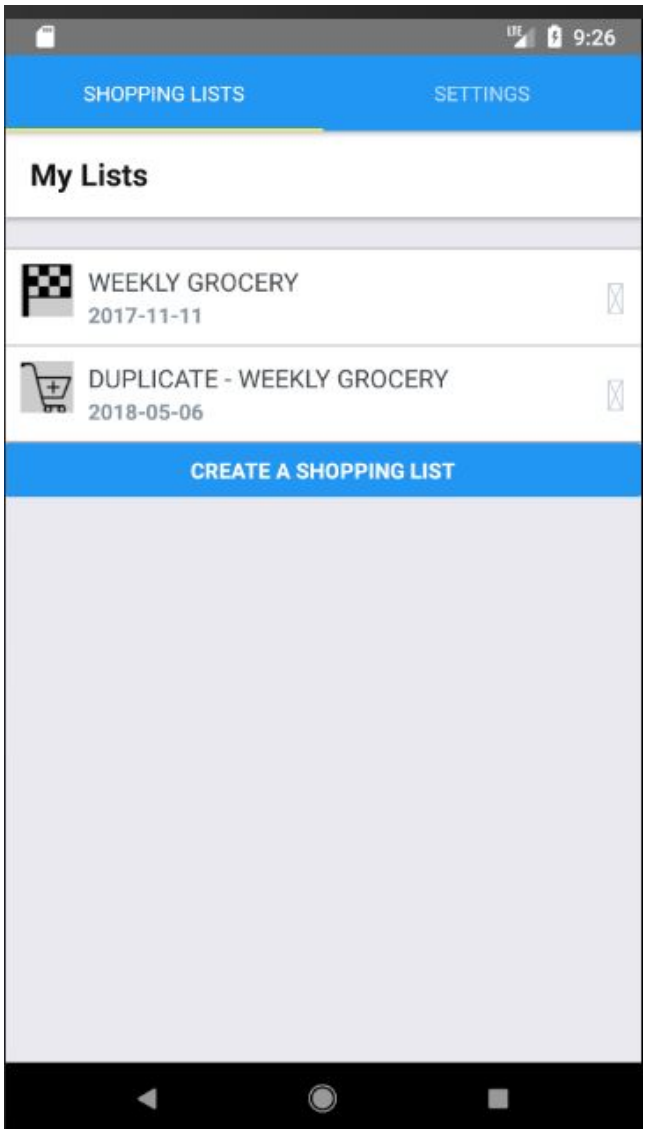



This component displays the user Image (or Shared Shopping List Icon), Nickname, Email and Notification allowance - whether the application will notify the user on changes at any of the user's active shopping lists.

Pressing edit will allow editing these attributes.

Figure 10 : UI Panels - Settings Page.

User’s Shopping Lists :

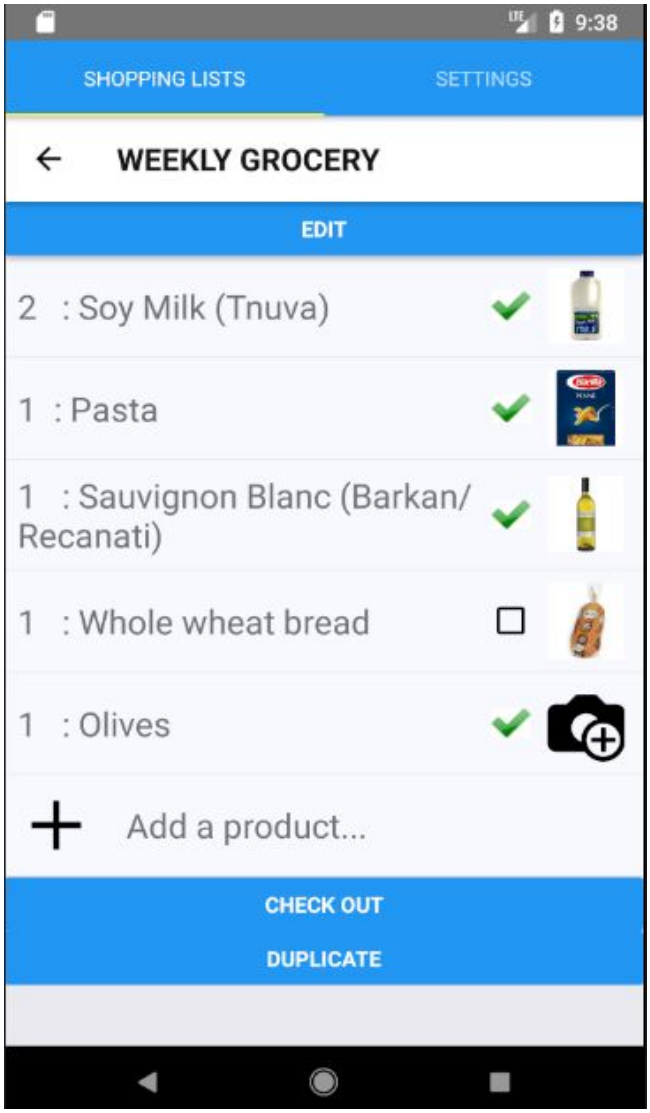



The Shopping Lists component displays the user’s Shopping List of which he participates.

Active shopping list will have a cart icon, suggesting that shopping is being or will be conducted. Finished shopping lists are deemed as non-active, and has a finish flag icon near them.

Figure 11 : UI Panels - User List.

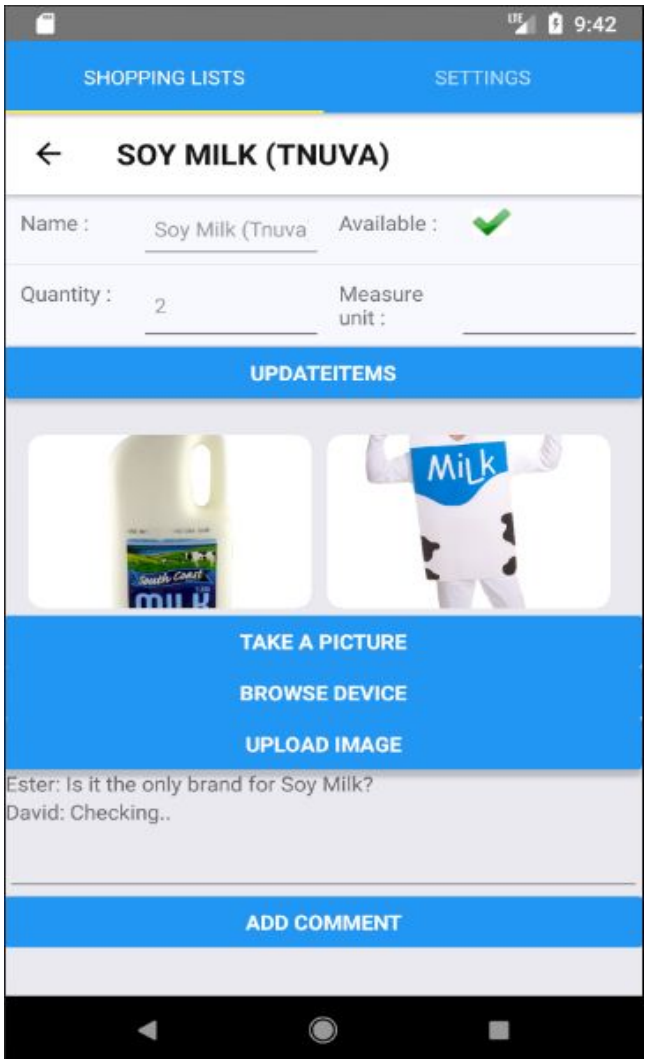
Shopping List Details :




This component has two configurations, Administrator and Participant. A participant could add product by press “+” and click on one of the items to see its’ details, and can duplicate this list with all its items to a new list, being the Administrator of the new duplicated list, and Administrator, can also mark item’s availability and check out when finishing the shopping.

Figure 12 : UI Panels - Shopping List Details.

Item Details :




Like in “Shopping List Details”, this component also has Administrator and Participant configurations, both can upload images by either taking a picture with the mobile camera and or browsing device (both requires application access by device), and add comments about the product. Administrator can also update the item details, include its availability.

Figure 13 : UI Panels - Item Details.

Back End - Services and Web Methods

The back-end of “Shared Shopping List” contains one service - ShoppingListApi. ShoppingListApi contains eleven secured HTTPS/POST web services to support all the use cases of the application. All the requests and responses are in JSON formatted, and could be further

The ShoppingListApi service is divided by three concerns: Items Management, Shopping List Management, and Participant Management.

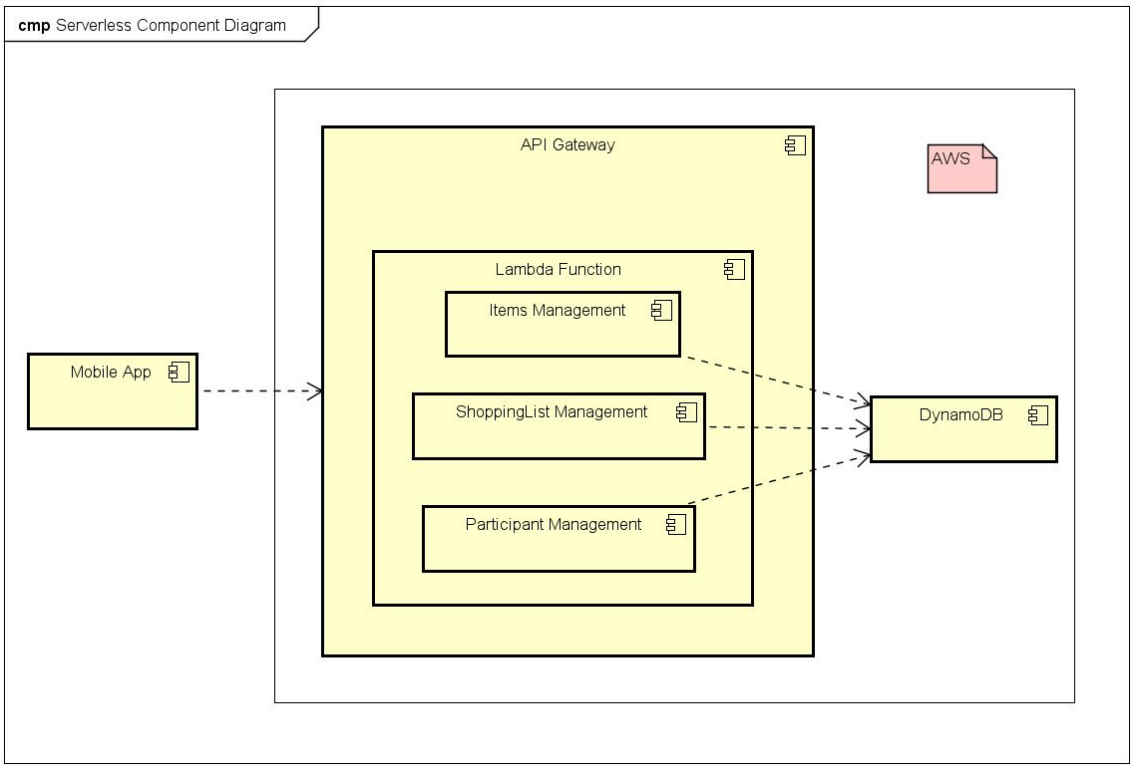



Figure 14 : Component Diagram - Serverless

A mobile application holds an address for the backend API Gateway. The application business logic deals with three main entities: participants, shopping list and items. Each has multiple functions which deployed over Lambda, released through API Gateway, and uses DynamoDB as a database.

## Items Management

Please note that the URLs examples are cannot be executed as they emphasis an example of what a request URL look like, and not the actual URL of a staging or production backend.

Append Item To Shopping List :

URL -/appenditemtoshoppinglist

Input - EditItemInListRequest.

Output - String.

Appends an item to an existing list, using EditItemInListRequest object.

EditItemInListRequest holds data about an item in a list, including the shopping list id, item name, availability, measurement unit and volume, URLs of image ids and comments about the item. Once the method AppendItemToShoppingList is executed, it will assert that the Shopping List exists, create and persist the new the new item, and append it to the list. While persist to the database occurs, a string GUID will be assigned to the ItemInList entity which has persisted in the database, this will help access it in the future. A

successful request will return the shopping list id, while an error will return an error message that occurred.

The following sequence diagram demonstrates the flow of updating an item in list :

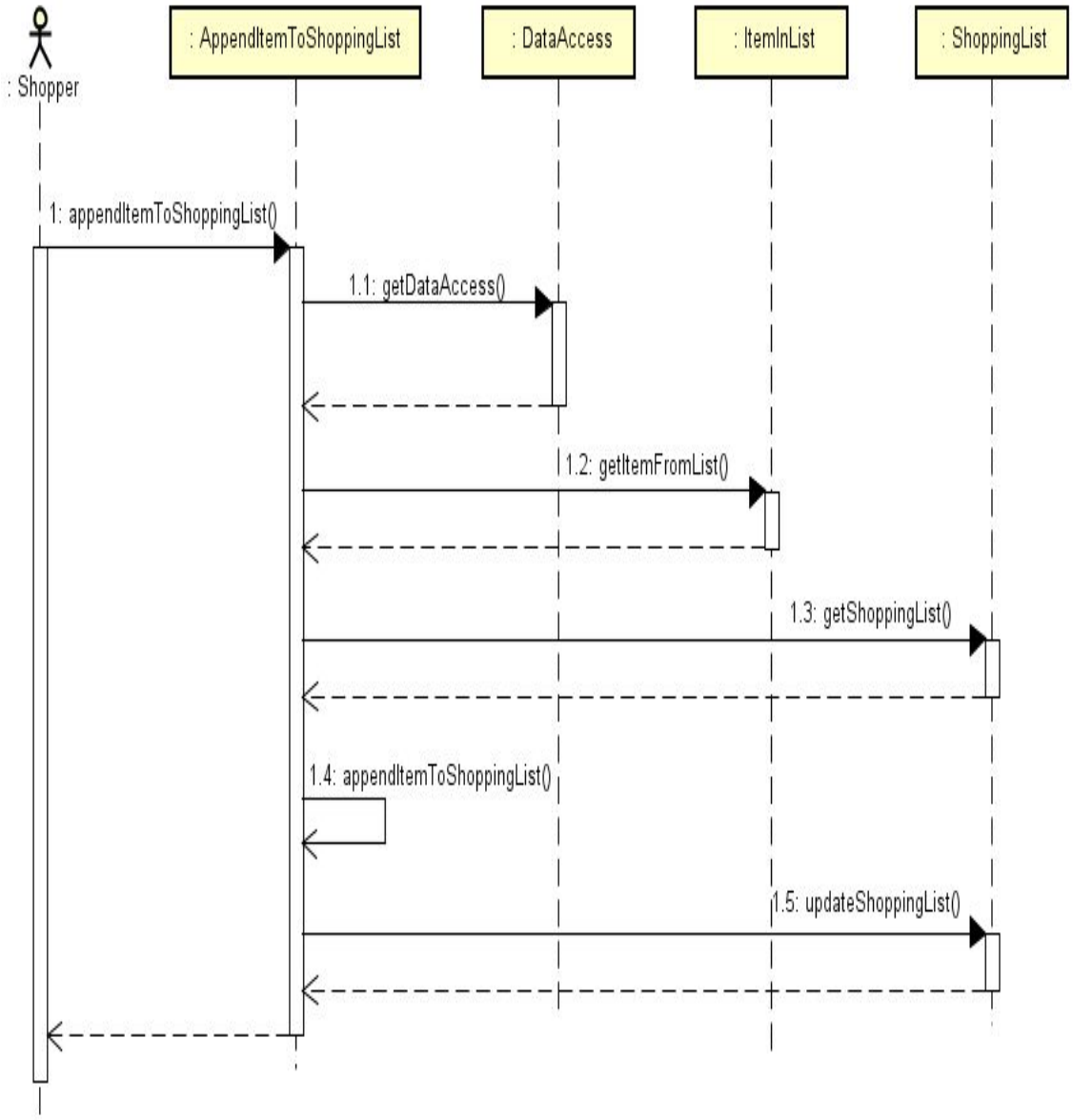



Figure 15 : Sequence Diagram - Append Item To Shopping List.

A user initiates an AppendItemToShoppingList request. Using Singleton Pattern, the DataAccess object should provide an instance of DynamoDBMapper, which persist the new ItemInList entity to the database, using the credentials that were given via EditItemInListRequest. Then, it will load a ShoppingList entity using their IDs which and append this new item to the list and update the ShoppingList entity.

Update Item In List :

URL - /updateiteminlist

Input - EditItemInListRequest.

Output - String : Item In List ID or error message.

Similarly to Append Item To Shopping List, Update Item In List receives an EditItemInListRequest, which holds and ID of the of the Item In List, along with the data about the item to update. Once the method UpdateItemInList is executed, it will check whether the Item In List exists using it's ID, and update it with the values from EditInListRequestItem. A successful request will return the ID of the Item In List, while an error will return an error message.

Participant Management

Get Participant By Id:

URL - /getparticipantbyid

Input - String : Participant ID

Output - GetParticipantResponse, includes participant details and error message.

Receives an ID of participant, and return details about this participant, including nickname, email, birthdate, list of shopping list ids, and notifications preferences, mobileId. Once the method GetParticipantById will be executed, it will check whether a participant with this ID exist and return the participant details or an error message.

Edit Participant:

URL - /editparticipant

Input - EditParticipantRequest

Output - GetParticipantResponse, including the participant details and error message.

Receives details about participant, including ID,  nickname, email, birthdate, list of shopping list ids, and notifications preferences & mobileId. When the method GetParticipantById is invoked, it will check whether a participant with this ID exist and update the participants details, and then return the updated participant details.

Get List Participant By Id:

URL - /getlistparticipantbyid

Input - String : Participant ID

Output - GetListByParticipantResponse, List of Shopping Lists and error message.

Receives an ID of participant, and return the details of all the shopping lists and items associated with this participant. Once the method GetListParticipantById will be executed, it will check whether a participant with this ID exist, get all the shopping lists which are associated with this participant, and all the items in lists for each of the participant's shopping lists. It will return a suitable error if no user exists with the given participant Id.

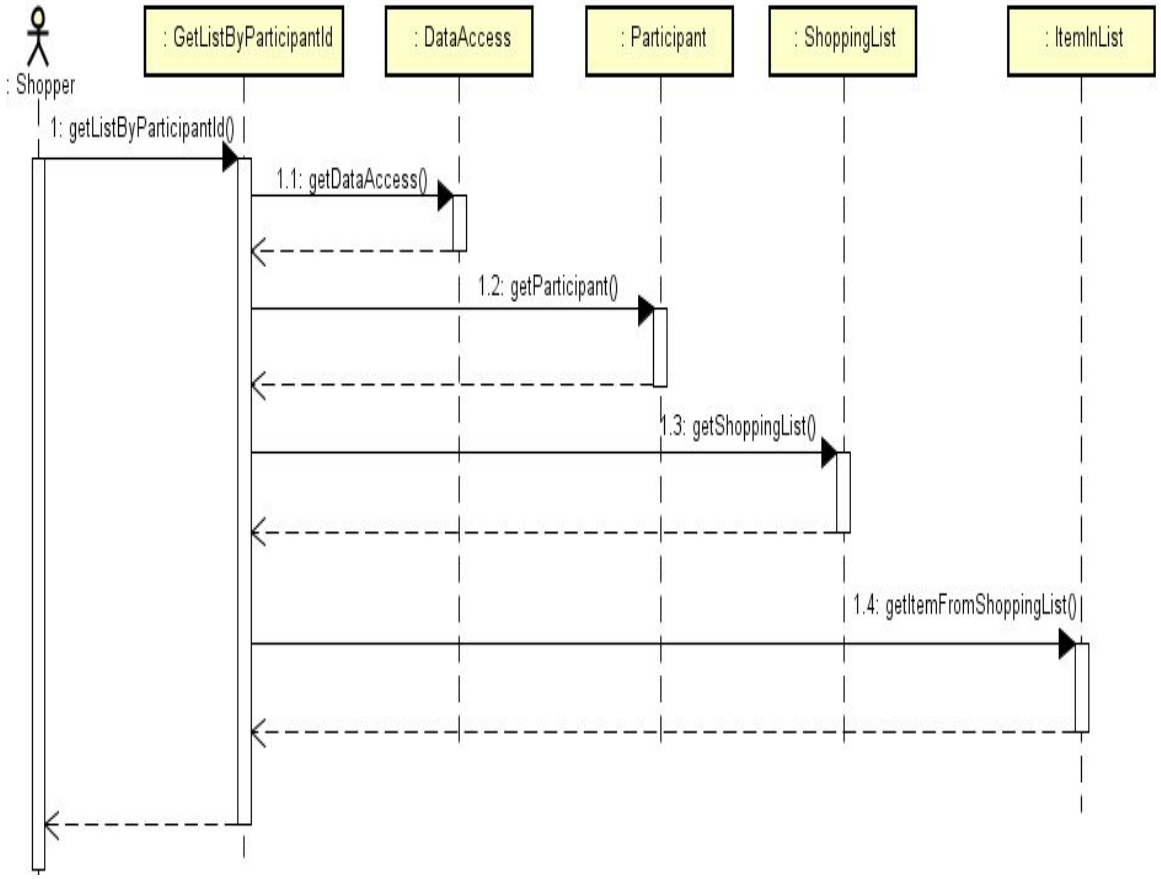



Figure 16 : Sequence Diagram - Get List By Participant ID.

A user initiates a GetListByParticipantId request. Using Singleton Pattern, the DataAccess object should provide an instance of DynamoDBMapper, which loads a Participant entity using the given ID of the request credentials. Then, fetching all of the ShoppingList entities that this user has, including the ItemInList entities of each ShoppingList, using the shopping list id of each of these ShoppingList entities.

Shopping List Management

Change Shopping List Administrator:

URL - /changeshoppinglistadministrator

Input - ChangeShoppingListAdminRequest: Includes CurrentAdminID, NewAdminID & ShoppingLisId

Output - String.

Receives an ID of an administrator, a participant and shopping ID, and changes the administrator rights of the given shopping list to the new participant, setting it to the administrator of the list. Once the method ChangeShoppingListAdministrator will be executed, it will check whether an old admin participant, new admin participant and shopping list with the given IDs exist, and will assign the old participant as the new admin of the list. It is possible that the new admin wasn't a participant at the list, in such case, she or he will automatically invited to the shopping list and become the administrator of the list. In any case, the old admin still remains a participant in the list.

ChangeShoppingListAdministrator will return an empty string for success, and an error message for failure.

Create Shopping List:

URL - /createshoppinglist

Input - CreateListRequest, includes the shopper ID, participant IDs, name item IDs.

Output - String : ID of the new shopping list.

Receives a request to create a new shopping list, with a participant ID of the user that created the request. Once the method CreateShoppingList will be executed, it will assert if the participant with this ID exist, if it does, it will create a new, active shopping list, with the participant as the administrator of the new list. In general, a new created list starts without any participants and items, these can be added later. CreateShoppingList Will return the ID of the new shopping list in for success, and an error message for failure.

Duplicate Existing List:

Url - /duplicateexistinglist

Input - DuplicateListRequest, includes the shopper ID, and original shopping list ID.

Output - String : ID of the new shopping list.

Receives a request to duplicate an existing shopping list, with a participant ID of the user that created the request. Once the method DuplicateExistingList will be executed, it will assert if the participant with this ID exist, if it does, using the ShoppingList object, it will create a clone of the same ShoppingList object using createClone method, which creates

and associates all of the items in list for the new clone, setting their availability status to "false", and invites all the participants of the original list to the new cloned list.

The name of the cloned list will be identical to the original one, and could be modified later via EditShoppingList. DuplicateExistingList Will return the ID of the new shopping list in for success, and an error message for failure.

The following sequence diagram demonstrates the flow of Duplicate Existing List :

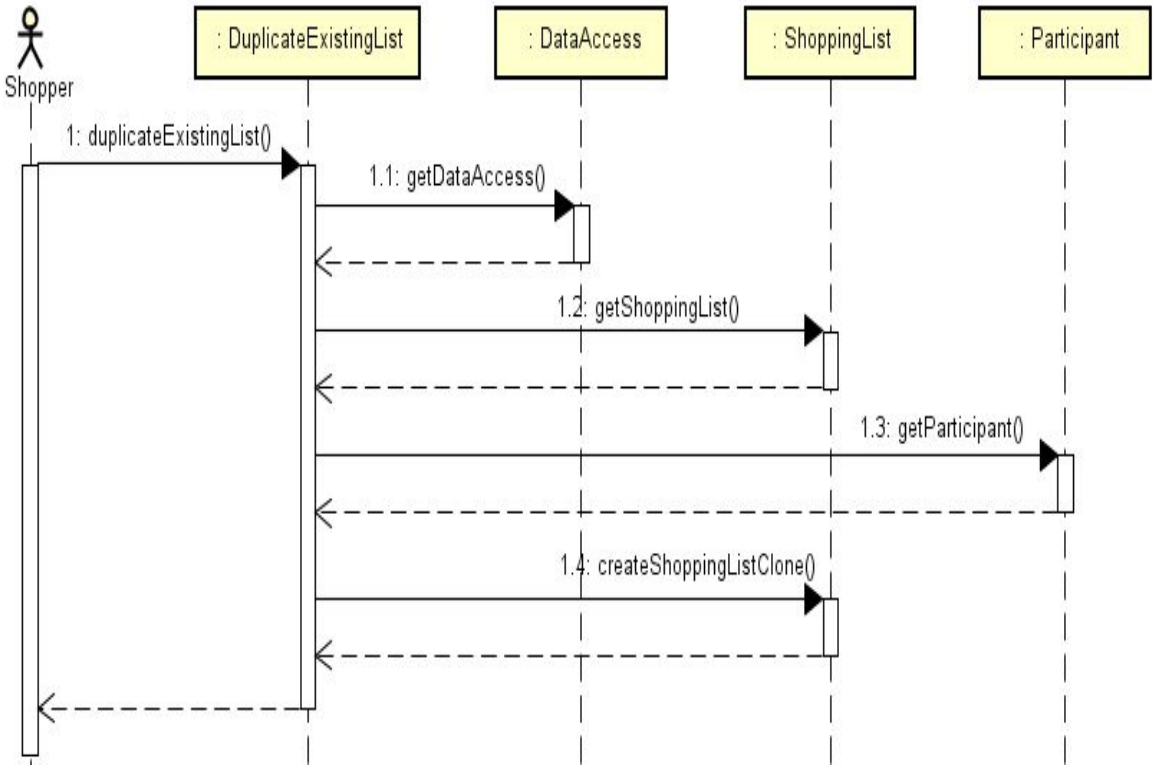



Figure 17 : Sequence Diagram - Duplicate Existing List.

A user initiates a Duplicate Existing List request. Using Singleton Pattern, the DataAccess object should provide an instance of DynamoDBMapper, which loads a ShoppingList and Participant entities using their IDs. Then, using the ShoppingList entity, cloneRequest method creates a new ShoppingList, sets it with all the relevant data from the previous one, sets it as active and assign the participant as the administrator of the new cloned shopping list, and persists the new shopping list.

Edit Shopping List:

URL - /editshoppinglist

Input - EditListRequest, includes the shopper ID, and shopping list ID, and list name.

Output - EditShoppingResponse: ID of the shopping list, or error message.

Receives a request to edit the name of an existing shopping list. Once the method EditShoppingList will be executed, it will assert that the shopping list id and participant with these IDs exist, and is the administrator of the list. If it does, it will update the name of the shopping list with the given name at the request.

Will return the ID of the new shopping list in for success, and an error message for failure.

Finish Shopping List:

URL - /finishshoppinglist

Input - FinishShoppingRequest, includes the shopper ID, and shopping list ID.

Output - EditShoppingResponse: ID of the shopping list, or error message.

Receives a request to finish the shopping of an existing shopping list. Once the method FinishShoppingList will be executed, it will assert that the shopping list id and participant with these IDs exist, and is the administrator of the list. if it does, it set the activity status of the shopping list to “false” and will set an end time of the list, using LocalDateTime helper. Will return the ID of the new shopping list in for success, and an error message for failure.

Get List By List Id:

URL - /GetListByListId

Input - String : Participant ID

Output - GetListByListIdResponse, includes Shopping List details with all of its related items and participant, and error message.

Receives an ID of a shopping list, and return details of the shopping list, along list of all of the shopping lists and items and participants at these lists. Once the method GetListByByListIdwill be executed, it will check whether a shopping list with this ID

exist, if it does, it will return all of the shopping list details such as starting time, ending time, activity status and more, including items and participants. It will return a suitable error if no user exists with the given participant Id.

## Application Utilities

Data Access:

This component is a wrapper class around DynamoDbMapper,an AWS DynamoDB connector library. DataAccess class uses the Singleton Pattern to create a single instance of DynamoDbMapper to provide CRUD operation of any entity which carries @DynamoDBTable attribute. Each database component has a class with a DynamoDBTable attribute, this allows DynamoDBMapper to fetch and persist any class with this attribute. Any DB component class, such as Participant, ShoppingList & ItemInList will have @DynamoDBHashKey & @DynamoDBAutoGeneratedKey attributes for it's ID GUID field Get method, and @DynamoDBAttribute attribute for any Get method of a field which will be persisted to the DynamoDB database. As stated previously, DynamoDB is a NoSql database, and as such, duplication of data could occur. For example - When a participant clones a new list, each item of the originating list is created again for the cloned list with the same properties as the original item.

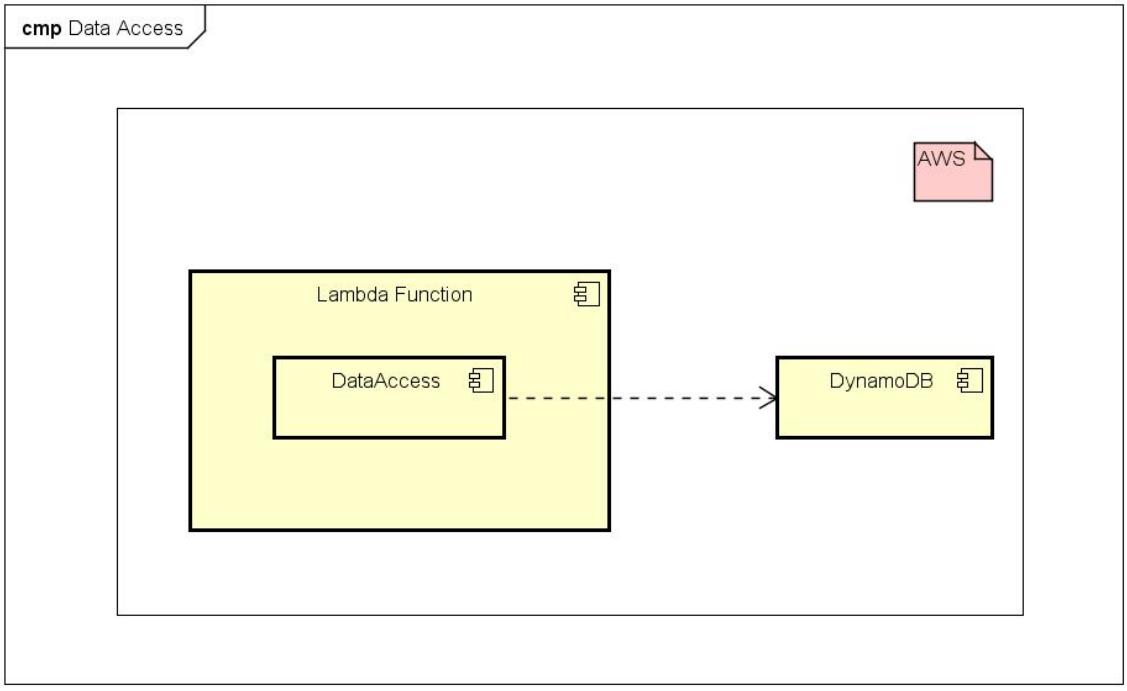



Figure 18 : Component Diagram - Data Access.

A data access object exists within each Lambda Function, and connects to the DynamoDB to perform any CRUD operation.

Request Handler :

Provided by AWS Lambda SDK, RequestHandler is a class which provides implementation of Lambda Functions. In Java, once declaring a class which will represent a Lambda Function, it will implement the RequestHandler class, with two generic types, in - for the request, and out for the response. RequestHandler will enforce implementation of a method called “handleRequest” which returns “out” and accepts two arguments, “in” and Context. Context is an AWS Lambda Utility which assists on logging and profiling the execution of the Lambda Function. Once deploying the Lambda Function, the AWS Console will ask for a definition of the Lambda Function, this definition, for Java, will be the package and class of the lambda function.

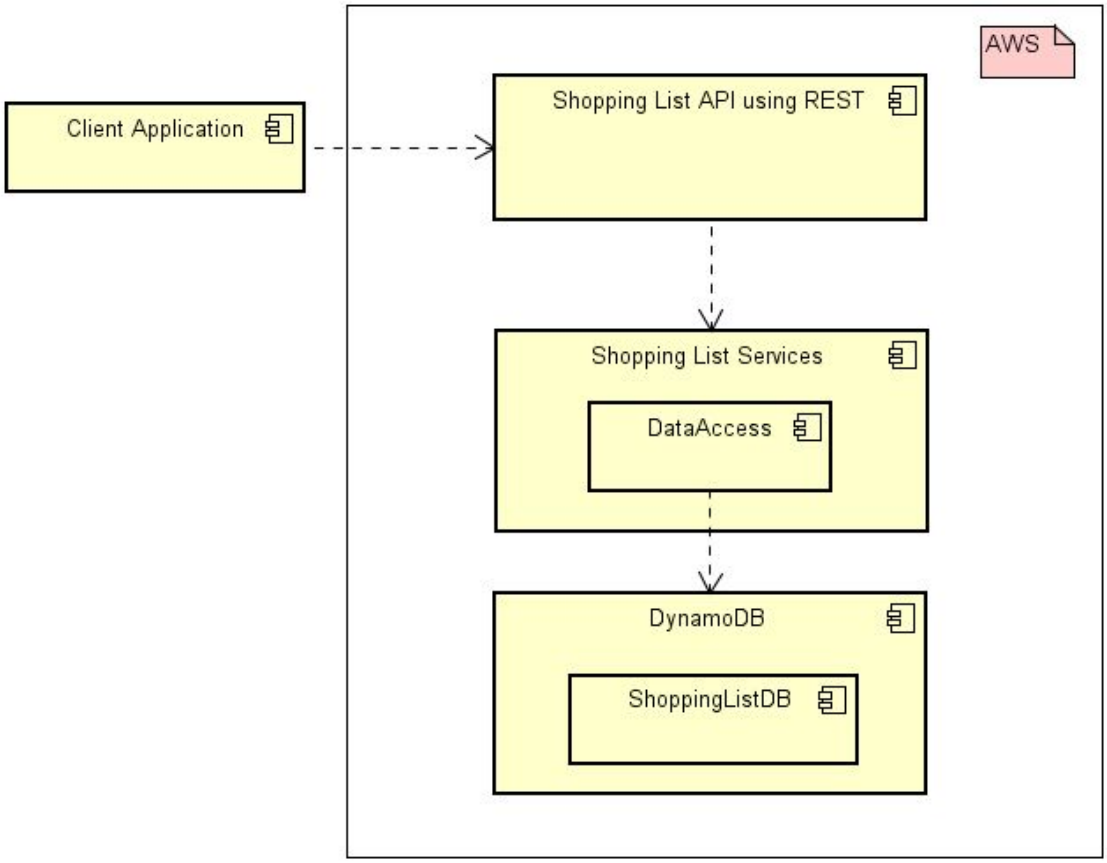



Figure 18: Component Diagram - Lambda.

Each mobile client is aware of one remote address: the address of the ShoppingList API gateway, which could be either staging for testing, or production for live. When a web method is invoked by the client, the API Gateway will execute a suitable lambda method to perform the desired operation, including an access to any CRUD operation using the DataAccess object. The AWS AIM insures that only requests from API Gateway will be able to invoke Lambda Functions, and only Lambda Functions will be authorized to perform CRUD operations on the DynamoDB database.

# Results and Evaluation

## Design Choices

### React-Native:

As stated previously, the project was supposed to be written using Ionic framework, and while working on the client application, a transition has been made to work with React-Native. Working with React-Native, understanding the framework, with emphasis on state management was probably the most challenging aspect of this project, the learning curve was more challenging than any other new technology, and understand how to properly design, write, and debug a React-Native, along with integration of new packages was significantly more demanding and time consuming than other parts of this project. Architecture wise, React-Native relies on background processes that interact with the native platform of the mobile device, as opposed to DOM manipulation of native web viewers. Yet, React-Native proved to be a good mobile development platform to create hybrid mobile applications.

### DynamoDb:

As with React-Native, the suggested database at the thesis proposal was RDS, an AWS relational database. To support scale and fast responses, a decision has been made to use a NoSql database, DynamoDB NoSql sounded like an interesting decision, both because it

is ready and available as an AWS service, and a free tier program that made it considerably cheap to use. During the testing and application tryouts, it seems that DynamoDB could be costly, and a monthly cost of a single user which writes and reads regularly to the DynamoDB through the application will roughly cost 8 USD a month.

## Application usefulness

At late stages of the application development, it was introduced to the thesis director, along with family, friends and colleagues of the thesis author for review and feedback:

1. Settings :
    a. Checkbox for “notifications” should be obsolete, use swipe instead.
2. User List :
    a. There is no option to change the name of a shopping list.
    b. Ability to duplicate list.
3. Shopping List Details :
    a. Looks too tabloid, should change to list, measurement units.
    b. Bigger fonts.
4. Item Details :
    a. Option to leave comments for an item.
    b. Merge Comments and Item Details to one screen, Item Details.

Through the implementation phase, all of the above suggestions and feedback suggestions were implemented.

## Learnings

For real testing proposes, some shopping sessions have been conducted using the application. During these sessions, the users were able to create products while the list is active, leave comments, update their user settings, the shopper could update an item and finish a shopping session successfully.

Both Lambda Serverless and DynamoDb were found as reliable technologies which respond fast to requests which have multiple fetch and updates from the database. For example, Get Lists By ParticipantId, a web method which gets multiple list and items from the database, respond within 170ms. In addition, there have never been a record of any crush or unexpected response from either Lambda or DynamoDb.

Agile development supports adjustment if feature priority and scope. Technology was used to create a scalable and extensible application on which additional functionality can be added efficiently.

The trade-off between fast implementation and exploration of technologies was one of the major concerns. To promote and enhance more learning and technology discovery, the author chose to focus and give more attention to design and technology learning than on

product issues. In result, the solution involved better understanding of the trade-offs of the technology choices, and how to implement them properly.

# Summary and Conclusions

## Features

Postponed :

Item Suggestions - Originally, when a participant wished to add an item to a shopping list, a suggestion service would look for the participant previous items and attempt to guess what was the desired item, or alternatively suggest an item by existing list of items. Developing a suggestion mechanism is a broad topic itself, and author wished to stay focus on the collaborative aspects of the application.

Alternative items - To prevent Shopping List Details UI page to hold more data than it already has on items, an option for alternative items has been removed.

Authentication - React-Native doesn't support a proper library which integrates with Google authentication and Android Development Tools.

Image Hosting - Instead of storing item images on services such as AWS - S3, the image links are updated to DynamoDB.

Unplanned :

Clone Shopping List : Naturally, users will use the same shopping lists over and over again, with small changes over time, a support has been added to application, to allow duplication of existing shopping list.

Edit Items : Because alternative items option is no longer available, existing items can be renamed and changed measurement units.

Comments : To have a better experience of an item change, and to prevent texting and calling regarding modifications, each item in a list has a comment section.

## Data Model

The following tables has been removed :

Address - there is no need to hold the address of the place where the shopping is being conducted. Should the future feature of the application will need data about shopping location, then it could be gathered using the embedded mobile GPS.

Notification Logs & Activity Logs - Logs could be used, access and monitored via AWS CloudWatch, the back-end functionality which handles logs could wrap the usage of log mechanism to other providers.

## Commercialize

There are two possible ways to commercialize this application:

1. Selling Data Trends. Because this application holds data on any item, purchased from any user, and any participant, this application has a great potential on tracking and detecting trends of shoppers and their dependents (participants), without disclosing personal data about a specific user.
2. Ads - usage of embedded advertisement component could generate money for both the application owner and the advertise company.

## Accomplishment

This project created a fully functional Shopping List mobile application, using the latest front-end mobile and back-end technologies, and discovered a new methods and software engineering paradigms for developing a scalable multi-user application. This project implementation proved that serverless architecture can be implemented as a back-end service for collaborative mobile applications, and can support growth of usage and scale. Scenarios such as creating and closing a shopping list, along with item editing, participant management and changes with-in list works well both for a single or multiple users.

## Plans for the future

Implementation:

1. Technical:
    a. Investigate proper usage of DynamoDB, best practices, reduce DB calls per web request, throughput tuning.
    b. Consider Firebase technology notifications.
    c. Authentication mechanism
    d. Tokenization and RESTful security.
    e. Consider S3 to host images instead of passing existing URLs.
2. Functional:
    a. Allow more than one administrator per shopping list, or allow roles.
    b. Suggestion mechanism.
    c. Invite users from mobile contact list.
3. Release:
    a. IOS version.
    b. Landscape support.
    c. RTL support.
    d. Non-Latin languages, Chinese, Hebrew, Arabic, Hindu, and more.

Documentation:

1. Publish a blog or participate with other courses at Harvard and beyond such as CSCI E39B, to contribute for the React-Native community.
2. Youtube video about the application.

# Appendix.

## Payload examples

Append Item To Shopping List


```
{
  "measurementVolume": "4",
  "measurementUnit": "#",
  "itemName": "Ketchop",
  "available": "false",
  "listId": "8426976b-ac2e-45b4-922a-3f86c1352abd"
}
```


Update Item In List


```
{
  "measurementVolume": "4",
  "measurementUnit": "#",
  "itemName": "Ketchop",
  "available": "false",
  "listId": "8426976b-ac2e-45b4-922a-3f86c1352abd"
}
```

Get Participant By Id


```
{
  "10f53f45-d49d-437c-bf36-fa51b87bd34d"
}
```


Edit Participant


```
{
 "participantId": "10f53f45-d49d-437c-bf36-fa51b87bd34d",
 "nickName": "nickName",
 "email": "nick@name.com",
 "allowSettings": "true"
}
```


Get List Participant By Id


```
{
  "10f53f45-d49d-437c-bf36-fa51b87bd34d"
}
```

Change Shopping List Administrator

```
{
  "shoppingListId": "9c06a446-0a79-4cc0-9d07-027159035e4a",
  "oldAdminId": "dbd3e9c4-9751-4b00-8d61-3e7e35d43539",
  "newAdminId": "10f53f45-d49d-437c-bf36-fa51b87bd34d"
}
```

Create Shopping List

```
{
  "shopperId": "10f53f45-d49d-437c-bf36-fa51b87bd34d",
  "name": "Eric's ShoppingList",
  "participantsIds": [
          "10f53f45-d49d-437c-bf36-fa51b87bd34d"
  ]
}
```

Duplicate Existing List

```
{
  "shopperId": "10f53f45-d49d-437c-bf36-fa51b87bd34d",
  "originListId": "8426976b-ac2e-45b4-922a-3f86c1352abd",
  "newShoppingListName": "duplicate"
}
```

Edit Shopping List


```
{
  "shopperId": "10f53f45-d49d-437c-bf36-fa51b87bd34d",
  "shoppingListId": "8426976b-ac2e-45b4-922a-3f86c1352abd",
  "name": "Eric's ShoppingList",
  "participantsIds": [
          "10f53f45-d49d-437c-bf36-fa51b87bd34d"
  ]
}
```


Finish Shopping List


```
{
  "shopperId": "10f53f45-d49d-437c-bf36-fa51b87bd34d",
  "shoppingListId": "70ffce99-dea7-4be7-a7d4-f7df51dd075f"
}
```


Get List By List Id


```
{
   "8426976b-ac2e-45b4-922a-3f86c1352abd"
}
```

# References


Abbott, J. (2017, September, 3). AnyList too much? You might like Buy Me A Pie. Retrieved from http://www.thesweetsetup.com.

Android Studio Documentation.(2017, July, 9). Run apps on the Android Emulator. Retrieved from http://www.developer.android.

Apple XCode Documentation.(n.d.). XCode. Retrieved from http://www.developer.apple.com.

Barthel, S. (2018, January, 3) Monoliths, containers or serverless: what is the way to go? Retrieved from http://www.hackernoon.com.

Bookwalter, J. (2016, March, 30). AnyList review: Grocery app crosses off nearly everything on our shopping list. Retrieved from http://www.macworld.com.

Butler, B. (2016, April, 7). What is Amazon cloud's Lambda and why is it a big deal? Retrieved from http://www.networkworld.com.

Cui, Y.(2018, January, 17). I'm afraid you're thinking about AWS Lambda cold starts all wrong. Retrieved from http://www.medium.com.

Finnegan, M. (2018, April, 27). What is Trello? A guide to Atlassian's collaboration and work management tool. Retrieved from http://www.computerworld.com.

Fowler, M, (2015, June, 3). MonolithFirst. Retrieved from http://www.martinfowler.com.

Ganguly, R. (2018, April, 16). How to create a REST API in Java using DynamoDB and Serverless, Retrieved from http://www.serverless.com.

Gartenberg, C. (2018, April, 27). Google Tasks review: Still more to do. Retrieved from http://www.theverge.com.

Herron, D.(2018, March, 10). Full guide to developing REST API's with AWS API Gateway and AWS Lambda. Retrieved from http://www.blog.sourcerer.

Neves, G. (2017, April, 25). Keeping Functions Warm - How To Fix AWS Lambda Cold Start Issues. Retrieved from http://www.serverless.com.

Occhino, T.(2015, March, 26). React Native: Bringing modern web techniques to mobile. Retrieved from http://www.code.fb.

Person, J. (2018, April, 10). Time to Upgrade from GCM to FCM. Retrieved from http://www.blog.google.

Vogels, W. (2009, October, 26). Expanding the Cloud: The Amazon Relational Database Service (RDS). Retrieved from http://www.allthingsdistributed.com.

Vogels, W.(2012, January, 18). Amazon DynamoDB – a Fast and Scalable NoSQL Database Service Designed for Internet Scale Applications. Retrieved from http://www.allthingsdistributed.com.

Yaskevich, A.(2017, March, 17). Apache Cordova: Is there a future for it? Retrieved from http://www.scnsoft.com.