# Technological Feasibility

September 26th, 2017

**Team Name:**
Gnosis Solutions

**Team Members:**
Kalen Wood-Wardlow, Christopher Simcox, Thomas Back, Kristoffer Schindele

**Project Sponsor:**
Dr. Leverington

**Faculty Mentor:**
Dr. Leverington

# Table of Contents

## Introduction:

Gnosis Solutions is a small team of four developers working under Dr. Leverington, who serves as both a sponsor and mentor, to produce a new breed of grocery list application. There are many grocery list apps on the market already, however, none take advantage of an untapped data resource: the app users themselves. Dr. Leverington has tasked us with the problem of using crowd-sourced data as an asset to build a mobile grocery list app that provides a streamlined user interface, and leverages user data to improve the experience for all users.

In this document, we explore the challenges, our analysis of those challenges, and the decisions we have made to overcome them. We conclude with our plan to integrate each solution into a single cohesive application, and a visualization of the interaction between each component.

## Technological Challenges:

The following section describes the technical challenges that we will face in our project. Each heading will be followed by an in-depth explanation of the challenge as well as preliminary information on our options to overcome it.

### Platform choice

- Since we have free choice of which platform to use in developing this application, significant research is needed to determine the best possible platform. The most attractive options we have looked into so far include: React Native, Android Studio and Swift. Our primary factors for gauging the feasibility of a platform include: code reusability, maintainability, and the skillset of the team prior to development.

### Information storage

- We will need a way of storing the following types of information reliably:
  - Geolocational data for grocery stores
  - Grocery item data (price, availability, store location)
  - Grocery store data (floorplan, hours, specialization)

### Offline first storage

- We want user information to be saved to a central database, however, we do not want to hinder the user experience by requiring an internet connection to use the app. Thus, our storage system has to support offline first storage, a system that preserves user data while in offline mode, then syncs it to a central database when internet connection is restored.

**Data transfer**

- To effectively preserve user data, we will need a system of collecting and transferring data from individual user devices to the central database. Since the integrity of shared user data is so important, we must put a focus on the security, safety, and responsiveness of this system.

**Geolocation utilization**

- We will need to be able to store, retrieve and process geolocation data in a fashion that is reliable within the context of our mobile application.

## Technology Analysis:

While our challenges are common in mobile app development, it is vital to take each component seriously, and consider all of our options in detail. First and foremost is the front end platform, which will be the interface for the user experience. Second, a data storage platform, which must maintain an offline and online database with no change to the user's experience. This functionality includes the ability to store different data objects, such as: geolocation, item information, and store positioning information. Given that offline and online user data must be able synchronize seamlessly, an optimal platform for our project must have the ability to handle asynchronous data archiving. Such a data transfer process must be consistent and accurate. Finally, we must harness the mobile device's geolocation technology to provide maximum utility for the user. This geolocation technology must be supported by platform that interacts with the system consistently and reliably as well.

### **Platform Choice:**

We envision our application being used by shoppers either preparing to leave for, or while hiking throughout the store, it is crucial that the UX guides the user to quickly enter information and navigate the app. Although UX is mainly achieved through theory and user testing, we must choose a platform that is conducive to a fluid user experience. This can be achieved with either of the Android or IOS development environments, however, we desire our application to be multi-platform. Additionally, developing the application for both platforms will save the experience of having to create additional platform ports. While several multi platform developing options exist, we will focus on React Native, Apache Cordova, and Meteor.

**React Native**:

Based off of the Javascript library, React, React Native invokes the native rendering APIs in Objective-C and Java bringing real mobile UI components instead of webviews and gives it the look and feel of a native mobile application for the platform. The fact that it renders with native API calls gives React Native an edge over other cross platform development platforms. Other platforms render using webviews, which can lead to performance drawbacks, a characteristic we want to avoid while building a fluid user experience. React also works separately from the main UI thread, which results in higher performance. Most of the developers on our team have experience in web development, and React is based off familiar JavaScript and XML elements that were originally built to render HTML and CSS. The team's prior experience will reduce the time it will take to learn. Since Native React is Javascript, the application does not need to be rebuilt to see changes, only refreshed. This saves time on waiting for builds, and does not limit our choice in development environment.

One major drawback of React is its maturity. iOS support was released March 2015 and Android in September 2015. The community is still discovering "best practices", and the documentation has been noted for needing room for improvement. Additionally, some native API are not automatically supported, but can be implemented with manual configuration. Native React adds another layer to the project, and can create more difficulty in debugging, especially in interactions between the React and host platforms. Overall, React's most significant negative is the uncertainty that comes with learning a new technology.

**Apache Cordova:**

Cordova also allows for cross platform development and building native applications using HTML5. This creates a web application and uses tools provided by the Cordova team and the SDK from the mobile manufacturers to package the application into a native runtime container for the target platform. The application runs as a single webview object that allows the user to interact with it as a web application. Working with Cordova is appealing because it uses familiar web development tools such as HTML, CSS, and Javascript. The application can be constructed using standard web technologies, but Cordova also features specialized plugins through third-party developers and its own framework.

One of Cordova's drawbacks is its use of a webview instead of rendering native UI elements, which can lead to performance issues. Our research also indicates that plugins are often more hassle to integrate than they are worth. Developers have reported that they have to modify most plugins before they can be integrated into their current project.

**Meteor:**

Meteor is a cross-platform development option for mobile as a Node.js framework. Node.js is a non-blocking Javascript I/O model for scalable network web applications. Research indicates Meteor is similar to Cordova in that both use a webview, but Meteor simplifies the process by allowing you to create a project using Node's package manager. The package manager saves time by setting up the basic structure for the project for you.

Unfortunately, Meteor does not use native elements, and is mainly used for smaller quicker projects. This makes us wary of depending on Meteor as our main platform. Meteor's main selling point is that it can be used to quickly build web applications, but also requires knowledge of the Node.js MEAN stack, (mongo, express, angular, and Node). Given that our team has little experience with these specific tools, the time it would take to learn them poses a major obstacle to development.

**Chosen Approach**: **Native React**

We have decided to choose React Native as it offers higher performing and costs less to build a solution. React Native can render actual native elements of each host to save on build time, and we can avoid the extra work of modifying feature plugins that we would have to tackle with Cordova.

| **Platform/Capabilities** | **Native Elements** | **Familiar Language** | **Less costly build time?** |
|---|---|---|---|
| **Native React** | O(Host's UI) | O(JS) | O |
| **Apache Cordova** | X(webview) | O(web languages) | X |
| **Meteor** | X(webview) | O(JS) | X |

**Proving Feasibility:**

In order to prove Native React will be capable of providing our desired UX, we will prepare a demo showcasing examples of the UX elements that will be in our project. The examples will be set up to illustrate how Native React is most conducive to a fluid user experience. Examples may include the ability to switch between multiple screens and enter form information on each without performance lag. Additionally, the UX will be set up to visually guide the user through the app without explicit instruction.

## <u>Information Storage and Offline Storage and Data Transfer</u>

Our application will contain features that rely not only on crowd sourcing, but user specific data, as well. With this in mind, user data should be immediately accessible to them. Such data includes past shopping lists, visited stores, and intelligent suggestions the app derives from the user's data to provide a faster experience. Additionally, their information should be available to the crowd sourcing and data analytics processes that serve other users. Examples include: which items are located in which stores, and where they are located inside the store themselves. The three database platforms we are interested in are Realm.io, CouchbaseLite and MongoDB.

### Realm.io:

Realm is the most popular of the remote sync database options. It features a full local database stored on the device. This removes the latency inherent in traditional database models where calls and queries must be persisted over a network. For a mobile application dependent on wireless networks, this goes a long way in improving performance.

A second benefit of this model is that it supports offline use. It has the same fully capable database online as offline, and it syncs the data once a connection is reestablished. The database itself features objects that are reactive, so both iOS and Android can communicate with the objects. Each object lets you define a schema for its structure, and is noSQl. A global listener is installed to to track object changes from device side and instantly run server code in response to changes. Developers used to JSON formatted data in web development find this platform easy to pick up and implement. The endearing aspect of Realm is that is has a robust platform to handle data sync and transactions that ensures no inaccurate and uncurrent data.

Like React, Realm is still a relatively young software platform, and best practices for its use are still being developed. Since it is an Object server, it runs into some issues when accessed with PHP or other web query languages. Fortunately, Realm was developed to work well with Native React so this should not be a problem.

### CouchbaseLite:

Based on Couchbase, CouchbaseLite is a lighter version developed for compatibility with mobile applications. In typical NoSql fashion, Couchbase light features a flexible schema-defined JSON format. This is an attractive feature for our team based on our history with web development. CouchbaseLite also provides native APIs for each host, so it is easy to develop across multiple platforms. Like Realm, CouchbaseLite also features an offline sync platform that keeps the data locally as well as on the server.

One of CouchbaseLite's downsides is that querying is based off of a system in which you must build a "view" and add data to it. This is similar to the creation of an index for tables in a relation database. After this, in order to obtain the data you need to request the view with filters by keys. This is a process foreign to our team, and could be a waste to learn when we have more familiar options to choose from.

**MongoDB:**

MongoDB is another popular NoSQL database used for lightweight web and mobile applications. It features a JSON-like document model for objects defined through schemas. It is very easy to scale for this reason as well, allowing distribution of the DB across multiple machines. Server initialization and the query language are both very easy to learn, which cuts down on time our team spends learning new skills.

Unfortunately, MongoDB does not have offline sync capabilities, which means we would have to build these ourselves. This is a major shortcoming because there are free database options with this capability. On top of this, queries are not lightning fast, and exclude JOIN operations, making them less flexible. It also has no built in transaction support, another feature we would have to implement ourselves. Additionally, MongoDB is quite young and lacks in both documentation and best practice cases.

**Chosen Approach: Realm.io**

Realm has been praised for its ability to be paired with Native React without struggle, making it the best choice to pair with React as our front end. It fulfills our requirement for offline and online sync, and has a more robust and reliable platform for syncing data than the other options. Queries are extremely fast, as they are performed on the local database rather than the network. This allows us to perform data analytics for the user faster.

| **Platform/Capabilities** | **Offline Sync** | **Native Objects** | **Quick Querying** |
|---|---|---|---|
| **Realm.io** | O | O | O |
| **CouchbaseLite** | O | O | X |
| **MongoDB** | X | X | O |

**Proving Feasibility**:

In order to showcase what we want out of Realm, we will want to build a native application that communicates remotely with a Realm server. We will test sync capabilities by building a local database on the app that syncs with the same database on the server. We will use the front end for some basic data entry, and ensure it can query offline, as well as sync with the database once connection is reestablished.

**Geolocation Utilization**:

We have two tentative ideas for capitalizing on geolocation technology built into most mobile phones to make the experience better for the user. The first is being able to recognize a store location by having the user enter their current location while at the store. The system will then detect if and which store they are at based on their geolocation. The second utility, which we are reserving for a future feature, is the ability for the application to locate where the user is in the store, as well as locate the items they are looking for.This feature allows the user to quickly find a product by identifying the item's location relative to them on a grocery store floor plan. The API we have considered for this requirement include React's Geolocation API, and an open-source cross platform Geolocation Module that works with React as well.

**Native React Geolocation API**

This API extends a current web spec developed by Mozilla. The geolocation interface represents an object able to obtain the position of the device. This information can be passed to a web or mobile application. The iOS and Android platforms user different method call, which could lead to integration challenges. However, it is the standard for Native React, making it a top contender.

**Cross Platform Background Geolocation Module for React Native**

This is an open source API for Native React that is featured on their site, but is a working port of a similar module for Cordova. This is a battery conscious location management component that works for both iOS and Android without any special modifications. This would be a great plugin to use, but does not feature much documentation on their github. However, it should still be considered based on the positive feedback from developers that have used it.

**Chosen Approach: Native React Geolocation API**

This seems like the obvious choice as full documentation for the API is featured on Native React's website, and has a ton of support. Since Native React is still relatively young, we opted for a more reliable option over third party and open source plugins.

**Proving Feasibility**

We will build a small application with React Native that will output the host device's location accurately, and be able to store that location in the Realm database.

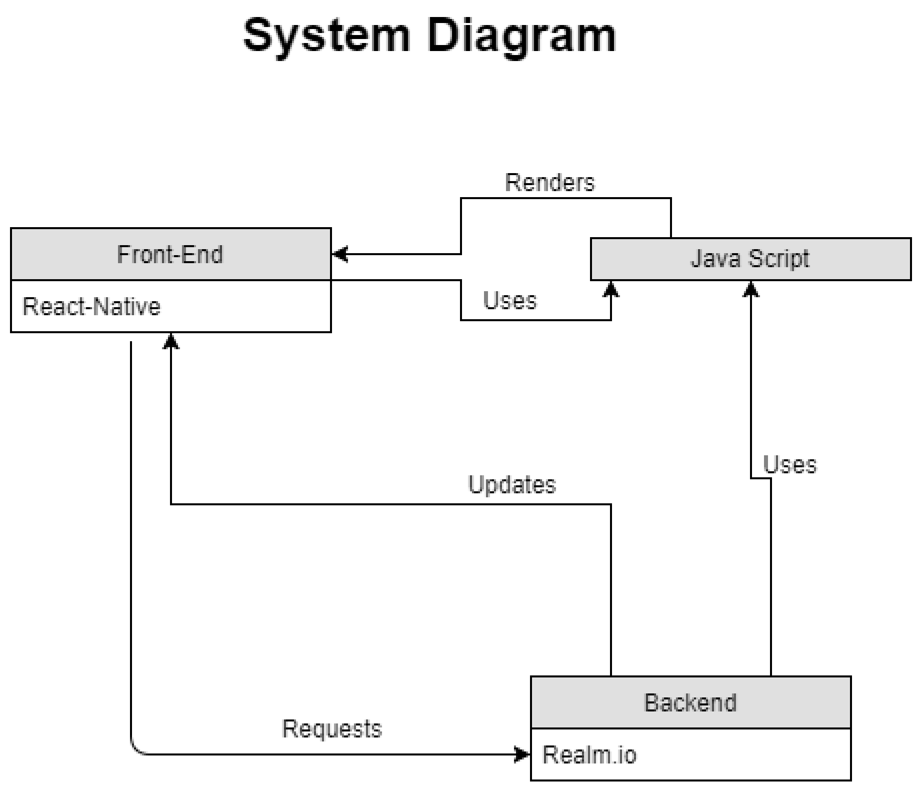
We will then need to query the data and compare it with new input to see if the locations match. This will prove that it will recognize locations and can store them in the database.

## Technology Integration:

We have decided to combine React Native as our multi-platform  front-end framework, Realm as our database platform, and React's Geolocational API to develop our application. Much of our choice was informed by the compatibility between these components, so integration should be relatively easy.

### Combining the Backend with the Frontend:

Realm.io provides a simple way to integrate itself into a react-native project through the Node Package Manager (NPM). Realm can then be imported into the javascript for the app, which has a library of functions for interface. Using this setup allows a simple way to integrate front end and the back end technologies using javascript as a medium for communication between them. This relationship is illustrated in the diagram below.



## Conclusion:

We believe that the combination of React Native, Realm, and React's Geolocational API provides a technology base most conducive to our design goals and requirements. The components were picked with an emphasis on compatibility. This allows us to spend less time integrating the technologies, and more time developing a streamlined UI and useful features.

A focus on the user's experience, and the asynchronous capabilities of Realm satisfy our client's requirements for a fluid user experience, and effective use of crowdsourced data. The Geolocational API will allow use to create features that leverage the crowdsourced data to further enhance the user experience. We are Gnosis Solutions, working under Dr. Leverington to build a mobile application that provides a uniquely optimized user experience through the use of crowdsourced and geolocational data.