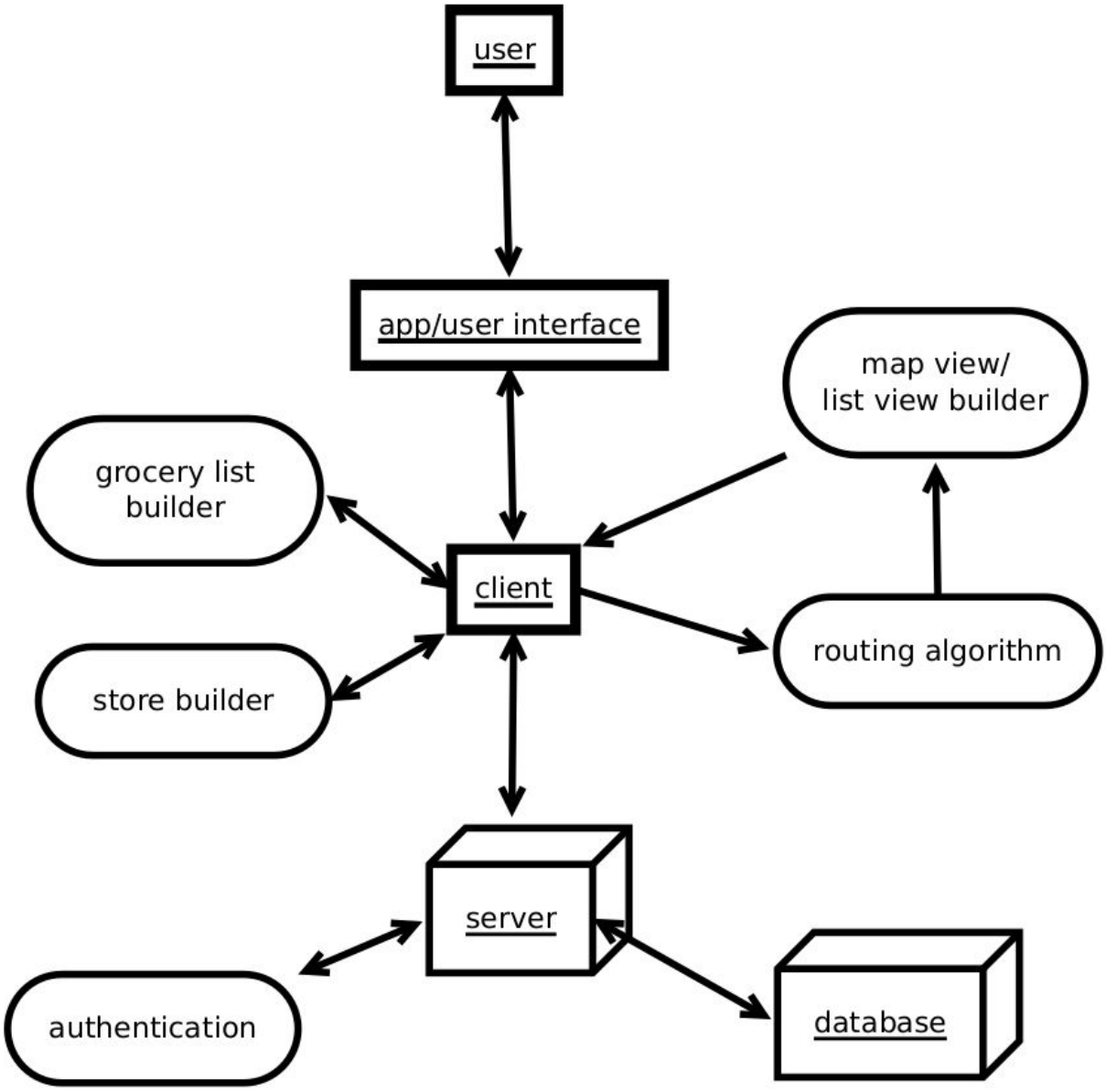
Grocery Shopping App Requirements
(Grocery Guide)

10/27/2019

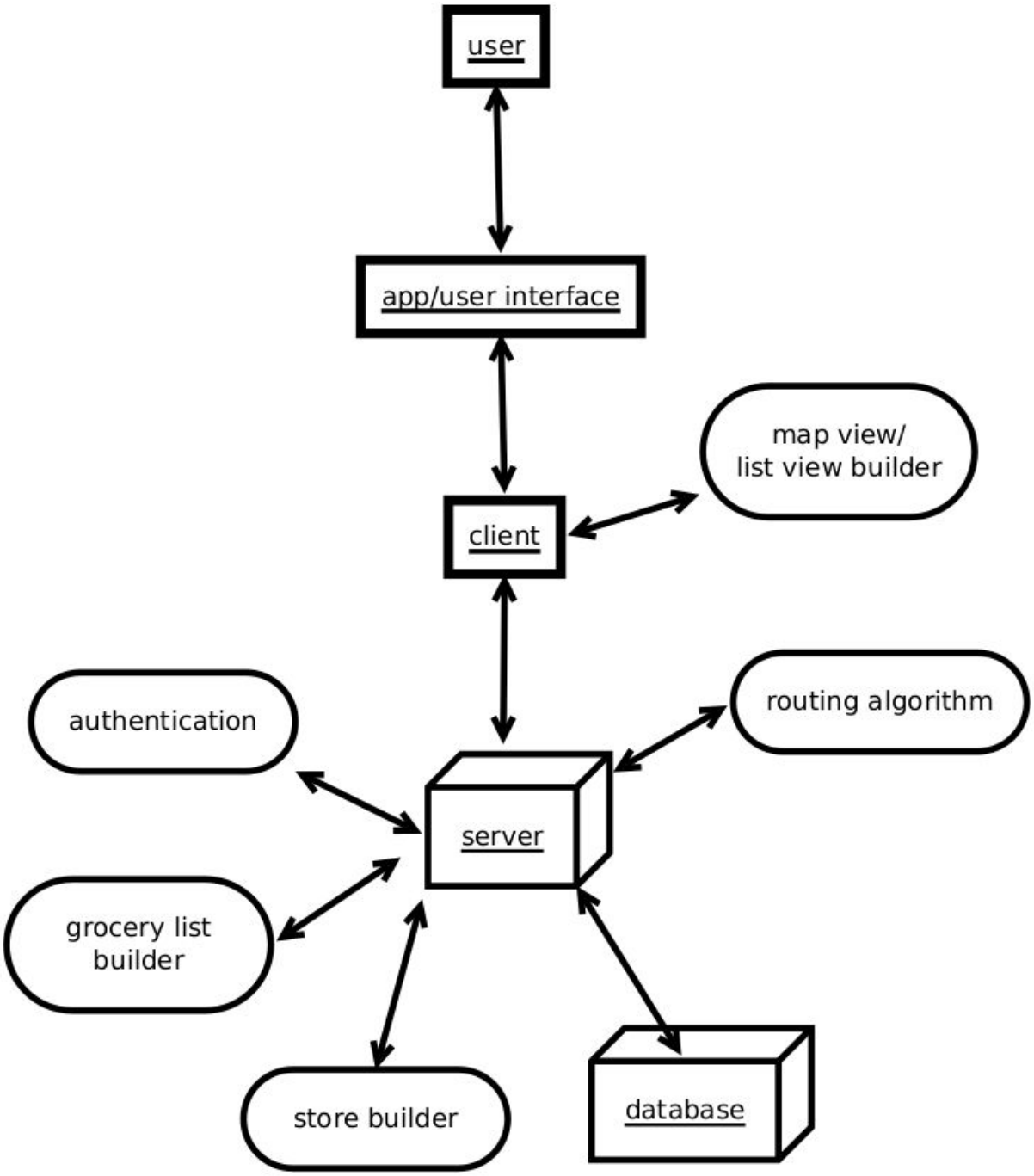
Christopher Brown
Dane Emmerson
Evan Horne
Anthony Huynh
Briana Willems

Data Flow Diagrams

Data Flow Diagram 1 (chosen architecture) - Client Heavy

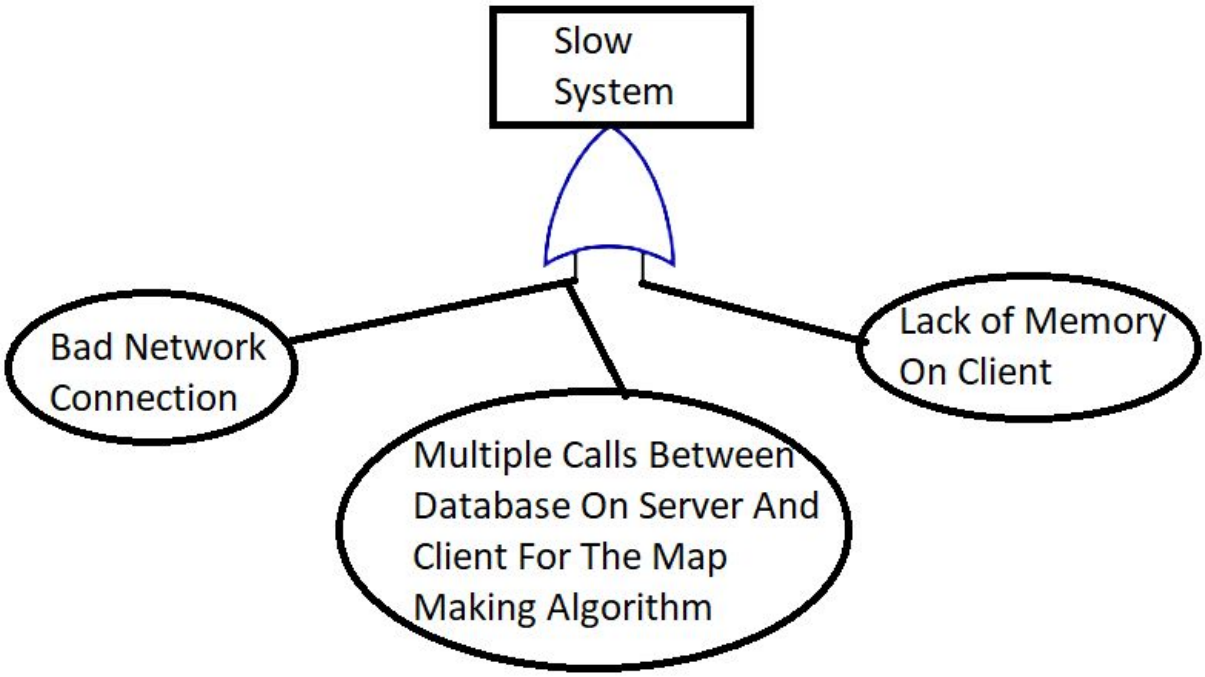


Data Flow Diagram 2 - Server Heavy

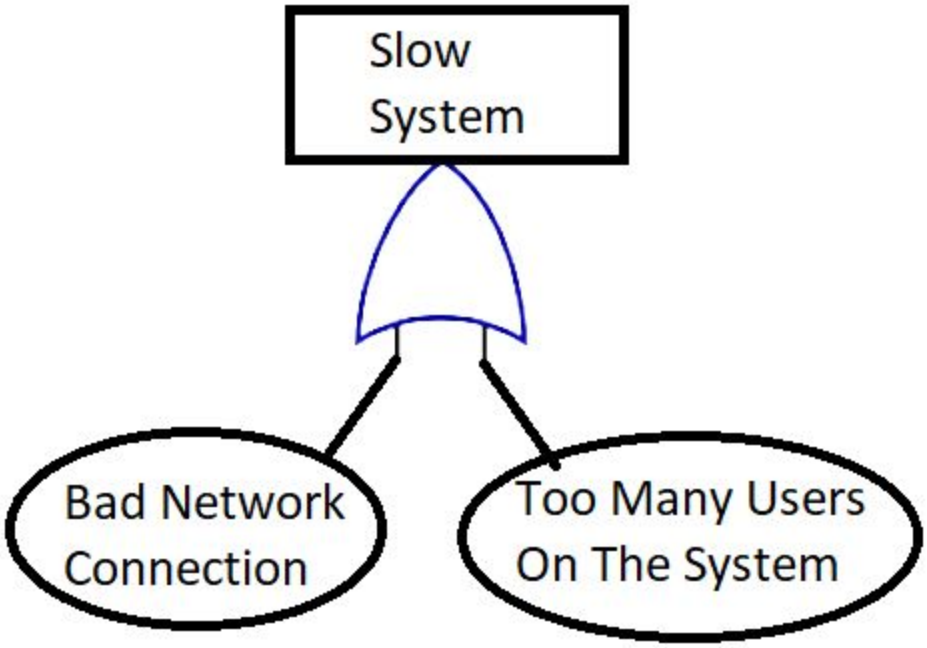


Fault Modes

Fault Mode 1- Client Heavy



Fault Mode 2- Server Heavy



Key Quality Attributes

Reliability

Architecture 1 (client-heavy)
The client-heavy design of architecture 1 provides a reliable model. The client makes minimal transactions with the server, thus removing negative effects of a slow internet connection and removing the need to be connected to the internet while using most of the app. Potential issues with this design could arise if the user is using the app with a slow device, or the user's device is severely limited with memory or disk space.

Architecture 2 (server-heavy)
The main pitfall to the reliability of architecture 2 would arise if the user's internet connection is slow, or the user has no internet connection at all. More specifically, it is not uncommon for cell service signals to be limited inside grocery stores. As a result, the user would be required to load their shopping their prior to entering a grocery store, and they would be unable to change settings such as picking up heavy items last.

Efficiency

Architecture 1 (client-heavy)
The design of the client-heavy architecture has the added benefit of relieving the load off our servers and infrastructure. A large portion of the app's calculations would be completed on the user's device, rather than tying up the server. However, this model assumes that the client's hardware will be sufficient to perform necessary calculations. With the current hardware of the majority of devices in use today (mobile phones and laptop/desktop computers), we do not foresee an issue with offloading calculations to the client.

Architecture 2 (server-heavy)
The server-heavy architecture design would require the servers to perform most calculations. The user would still be able to switch between a map and list view of their shopping list given the data flow of this architecture, but map building, grocery list building, etc would require a constant connection to the server. Calculations could be done quicker on the server than say a mobile device (without accounting for network transfer time), but adopting this architecture would force us to require additional server infrastructure to serve content.

Portability

Architecture 1 (client-heavy)
Portability is the biggest concern with the client-heavy architecture. Because a large portion of the calculations will be made on the client, and the client could be using any number of devices with our app, the majority of our app either will have to be written in a platform agnostic language, or we will have to deliver the app in a manner specific to users' devices - for example, creating mobile app versions as well as a web-based version for laptop/desktop use.

Architecture 2 (server-heavy)
The server-heavy architecture would be slightly simpler to implement from the standpoint of portability. One set of code, ran on the server, would run the majority of the app. However, in order to deliver this app to a wide audience, we would still need to provide a user interface that works across a wide variety of devices. Thus, we encounter the same issue as with the client-heavy architecture and there is not a huge portability benefit to the server-heavy architecture.

Testability

Architecture 1 (client-heavy)
Because the client-heavy architecture requires more code to run on a wide variety of users' devices, testability becomes more convoluted and difficult. We will need to thoroughly test our code across the various client platforms that our app supports and verify that our app's software runs as intended on various hardware. The increased focus on testability on this architecture will cause maintainability to be more cumbersome as well.

Architecture 2 (server-heavy)
The server-heavy architecture has a slight testability advantage relative to the client-heavy architecture. Because less code would be running on the client in this architecture, the server-heavy architecture is more easily testable. Testing our software would focus more heavily on our own infrastructure, with known hardware, and less on the client.

Maintainability

Architecture 1 (client-heavy)

The nature of the client-heavy architecture will require maintaining and updating a larger set of code across various programming languages to guarantee the user experience. Maintaining more code translates to more planning and higher maintenance costs. The client will need to be automatically updated at certain times to ensure the app performance. These updates will likely take place upon the user opening the app, with the user's permission of course.

Architecture 2 (server-heavy)

Running more of the apps codes on the server would make maintaining the software a little bit simpler. We would be able to make a number of changes to the software by updating just our servers. Many updates could be rolled out without the end-user having any knowledge or need to know that an update has occurred. The server-heavy architecture would ultimately allow for an easier to maintain system when compared to the client-heavy architecture.

Usability

Architecture 1 (client-heavy)

The client-heavy architecture shines in terms of usability. After the user has downloaded the app, the user will be able to use most features on the app with only occasional need for an internet connection, such as when the user would like to download a store map. Being able to use the app offline is extremely important to the user using the app while actually shopping as cell reception is often minimal inside stores. Additionally, since most calculations will be done on the user's device, a slow internet connection will not inhibit the user experience, while performing tasks such as building a grocery store floor plan.

Architecture 2 (server-heavy)

There are several potential issues with the server-heavy architecture and usability. First, the user would not be able to use the app offline. As a result, limited cell reception inside a store could negatively impact the user experience and ultimately discourage people from using the app. Additionally, with the server-heavy architecture, a constant internet connection would be required to use most of the app - build store maps, build grocery lists, change the shopping route preference, etc. A slow internet connection would disrupt the app performance for all these tasks.

Use Case Walkthroughs

Use Case #1 – Creating a list

This is a simple client-hosted and stored task regardless of chosen architecture. After the grocery item/store dictionary is downloaded from the server database, all further activities are performed on the user's device. Through the grocery list builder interface, the user begins typing out an item. A Levenshtein distance-type algorithm gives auto-complete suggestions, and when the user finally hits enter a lookup is performed on the dictionary using the string as a key, with the return value being the department the string should be displayed under. Hosting the grocery item/department dictionary on the client allows lists to be created without cell service, and could allow for modifications if the local grocery stores an item in an unexpected department. The user can then save the list on their device, or select a map from their local archives or pull a store map from the server database.

Use Case #2 – Building a map

In this use case, the user starts by creating a local map object on the client device. They fill out some store metadata such as name and address, then select a store size and click next. This will initialize a two-dimensional array on the client device based on store size, which is displayed as a blank grid on the app interface. The user can then select an established grocery department or feature from a drop-down list, which causes selection of a grid cell to be marked as that department. Specifically, the index of the array associated with that cell is assigned a value corresponding to the department selected on the dropdown. When the user is satisfied with their map they click to save, and an algorithm scans the array until it finds an entrance/exit, a cashier station, and at least 3 different departments. If this is unsuccessful, the client informs the user to fix the issue and returns to the map screen. Otherwise, the data is condensed on the client by packaging a map object without whitespace (associating a list of array indices with their departments) and stamped with metadata, then pushed to the server database. This causes less packet requests to send a full map from the server, giving better performance when downloading/uploading maps.

Use Case #3 – Picking items from a list

Here a user has a created list and is ready to shop. They select a store through their list view, and the store map is downloaded from the server if necessary. The client then "unpacks" the map data into a two-dimensional array stored while the user shops. An algorithm first splits the grocery list if the frozen or heavy foods last option has been selected, and then creates a

“blacklist” of those departments. The remaining list of items is then converted into a list of their associated departments, which is fed into the search algorithm. The search algorithm first finds the entrance/exit, then performs a breadth-first search algorithm until a desired department is found. That department’s items are then added to the queue holding data for the list view, and the map view path is updated. This process repeats until the first list is clear, then is performed from the current cell down the “blacklist” departments to be picked from last. Once all departments in both lists have been visited, a final breadth-first search to the cashier is performed. The list-view queue and map-view grid are then hosted on the client until shopping is completed.

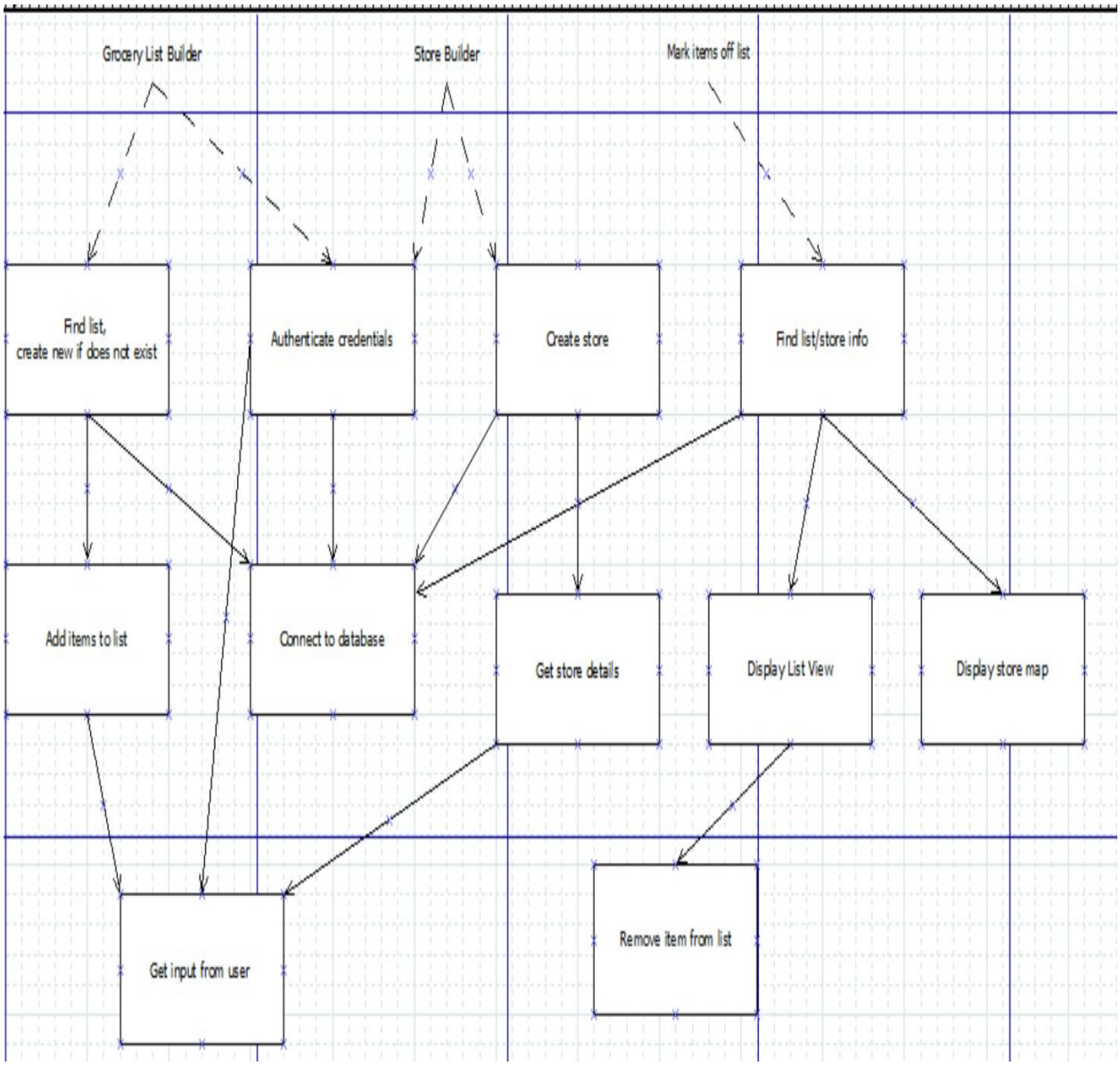
Implications of Client-Side Architecture

Choosing to move forward with development predominantly focused on client-side architecture eliminates significantly more opportunities for failure than server-heavy architecture. A client-heavy model makes very few design principle compromises as compared to a server-heavy model. As the current client-side architecture stands, the main issues making design adherence difficult are cellular/internet connection and device ecosystem support.

The current client-side architecture model maintains a lower dependence on cellular connection to the servers than its alternative. User's devices still have to pull updates from the server when store map and product updates are available, allowing some users the possibility of falling victim to slow/non-existent connection faults. Unfortunately, there isn't a single architecture design we can implement that does not require access to a server one way or another. Because of this, there are few revisions that can/need to be made in regards to internet connection. One possible addition could be the implementation of a store map/product update check that is kicked off upon the user tapping "Go Shopping".

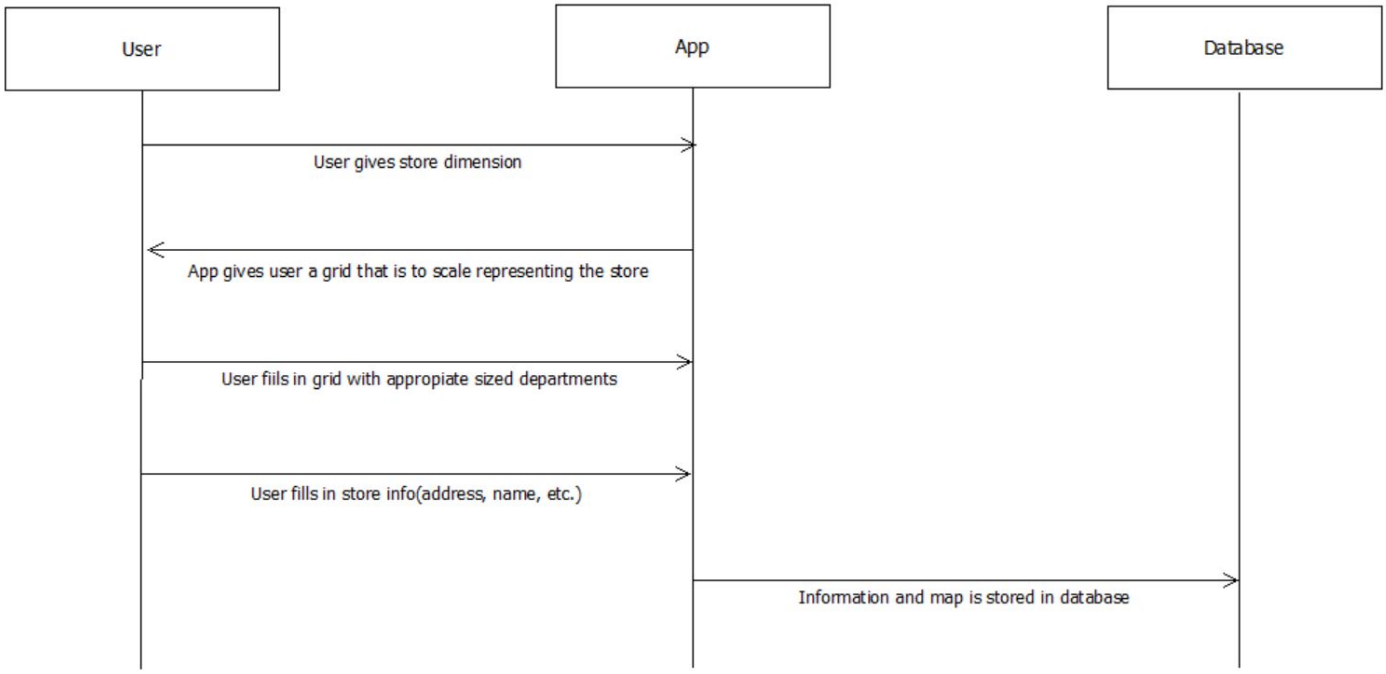
The other main complication to design adherence is device ecosystem support. There are dozens of different software platforms all supporting hundreds of different devices. Because of this, testing, code maintainability, version iteration, and ensuring our design principles are consistent across millions of devices can become quite difficult. One user with a certain phone, using certain screen dimensions, on a certain version of Android/iOS could have a completely different experience than a user on a well known flagship device. To summarize the relationship between device support and failure potential: As the number of devices supported goes up, so does the potential for failure. Taking a client-side development approach makes device ecosystem support a mandatory area of focus in regards to revision. One possible solution to ensuring support for many devices is to use a cross-platform app development framework like Adobe PhoneGap or Xaramin. These frameworks transpile code developed in a singular, unified language to multiple device platforms natively. However, this is less of a revision to the client-server architecture, and more of a development environment change. A controversial and "bigger picture" approach would be to limit app builds to only a certain library of well-known flagship devices. Doing so lowers the potential for failure due to the removal of obsolete and poorly designed phones, however this is a trade-off with availability.

Functional Decomposition

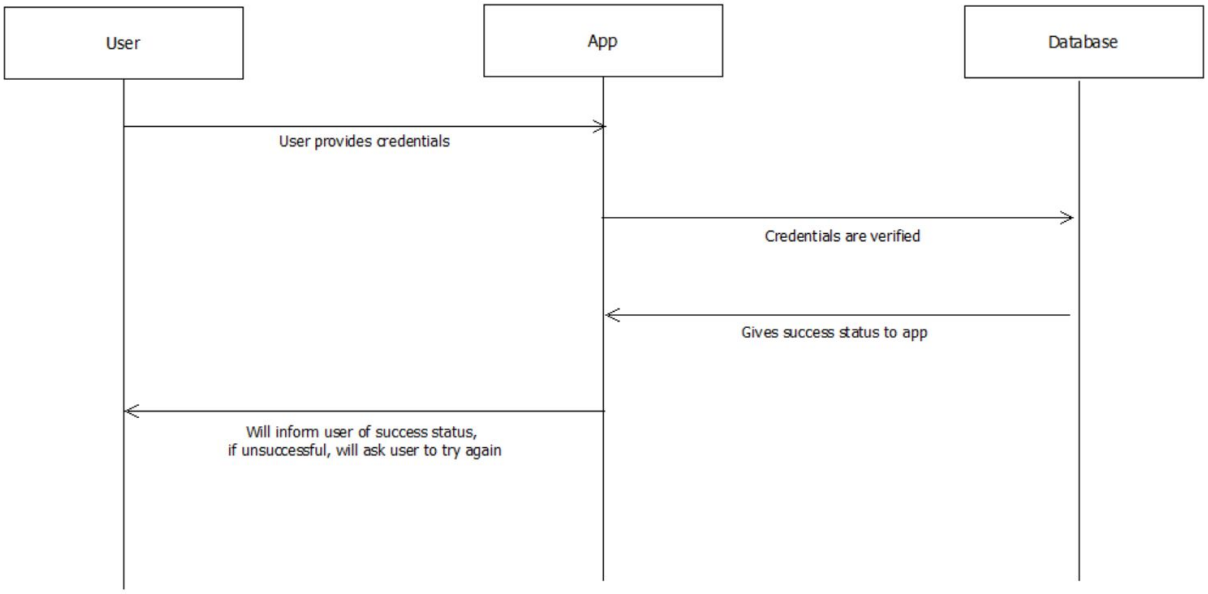


Data flow

Building store map



Logging in



Contributions:

High-level data flow diagrams: Dane Emmerson
Quality attributes: Dane Emmerson
Failure modes: Briana Willems
System decomposition and lower-level dataflow: Anthony Huynh
Validating selected architecture: Chris Brown
Explaining implications: Evan Horne