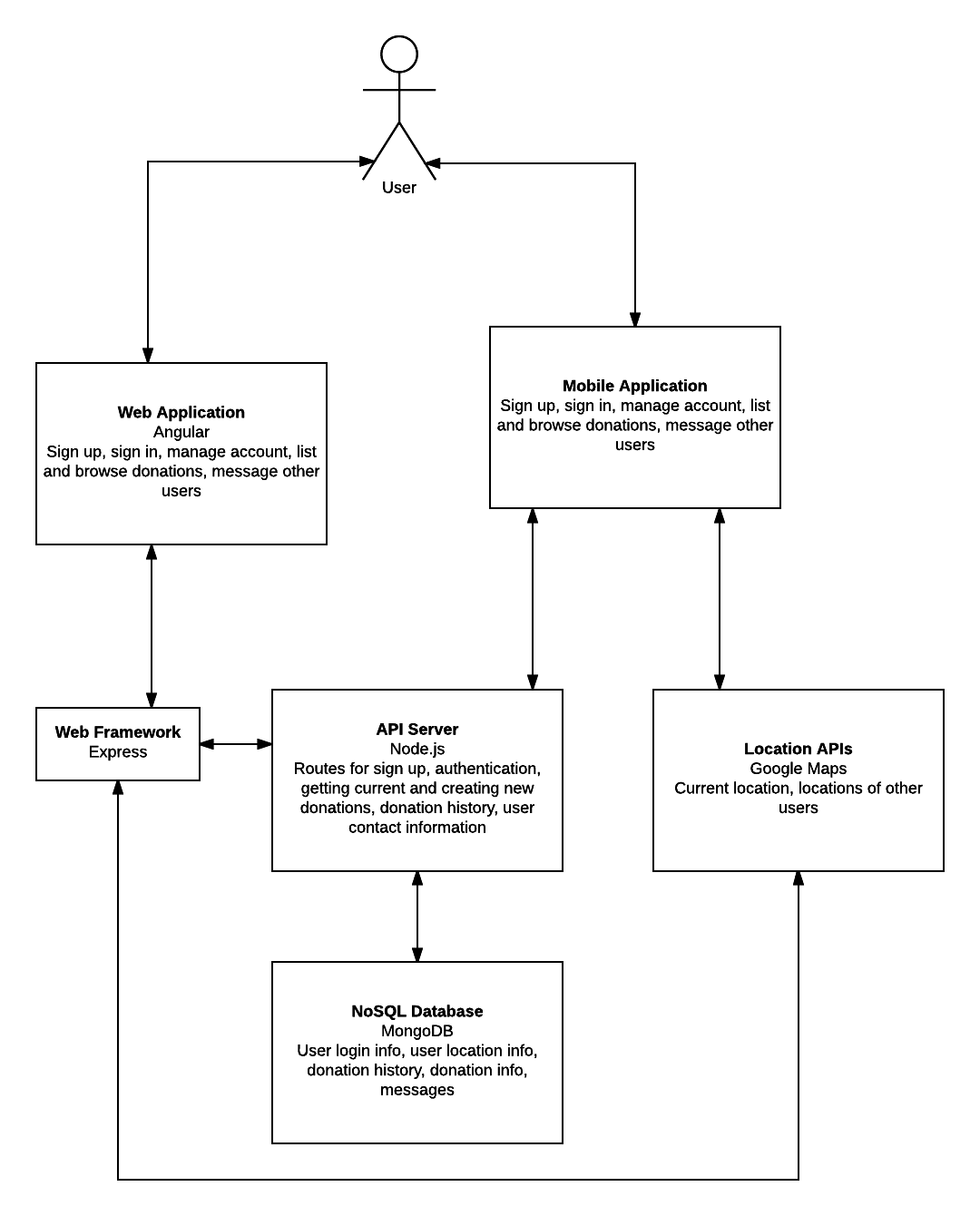
**Food Trucks Fight Hunger - Architecture Assignment**

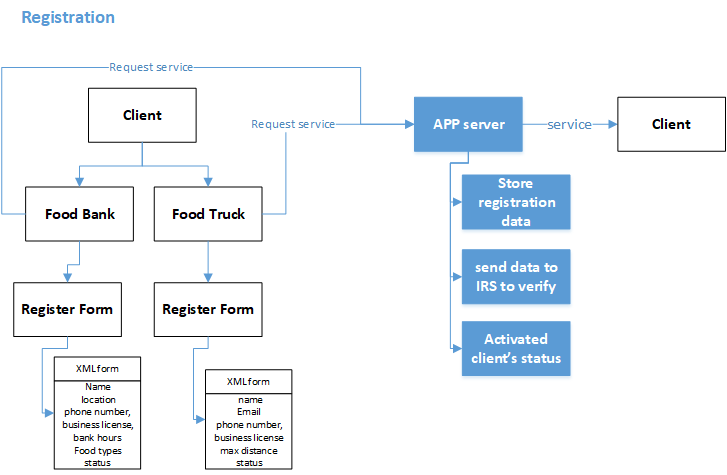
Group 24

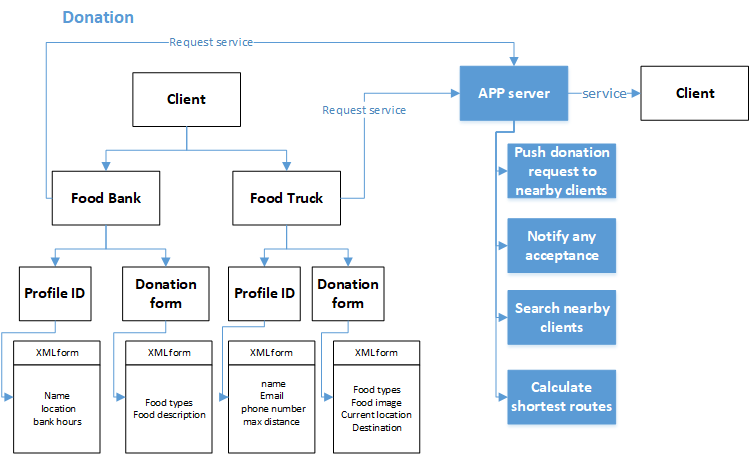
Antonio Bermudez  
Aaron Berns  
Wenwen Dong  
Steven Nowicki  
Hunter Schallhorn

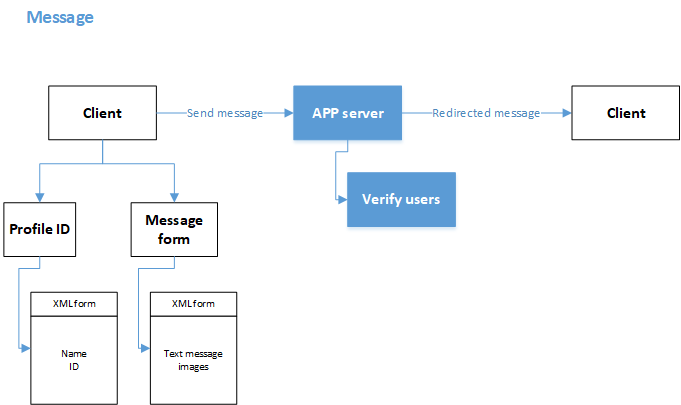
#### High-level architecture

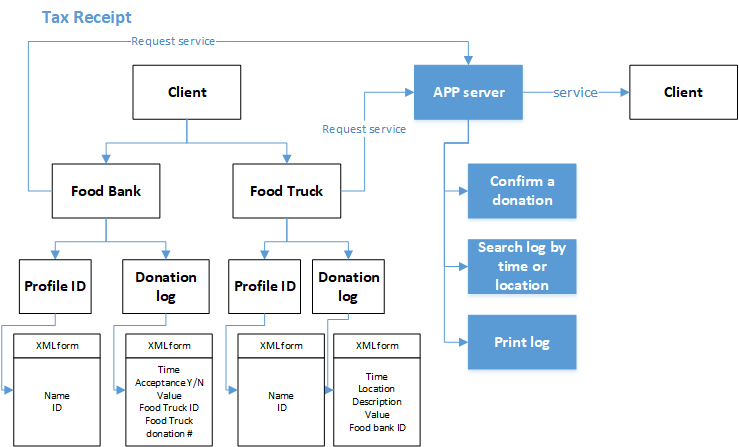


#### Alternative architecture



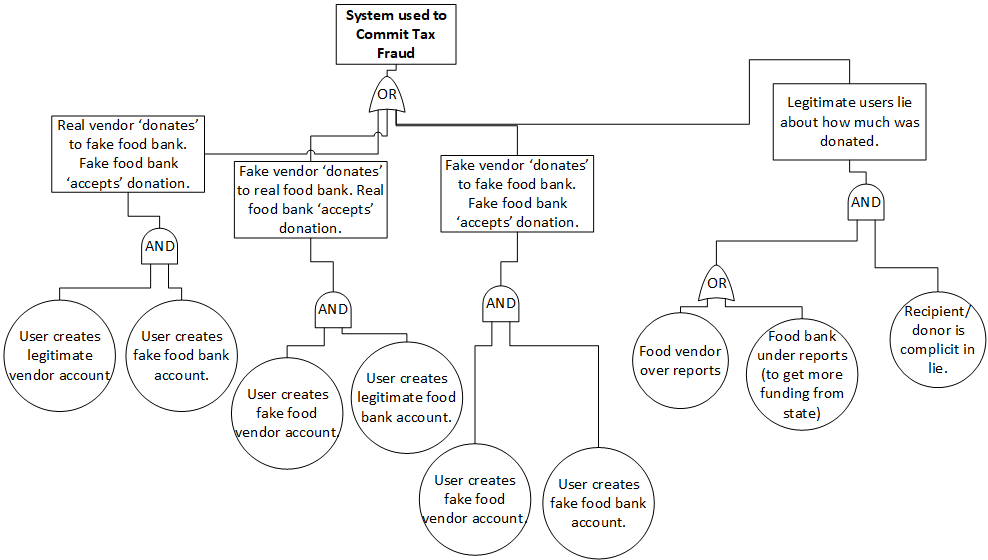






#### Failure Modes

Failure Mode 1: System Used to Commit Tax Fraud

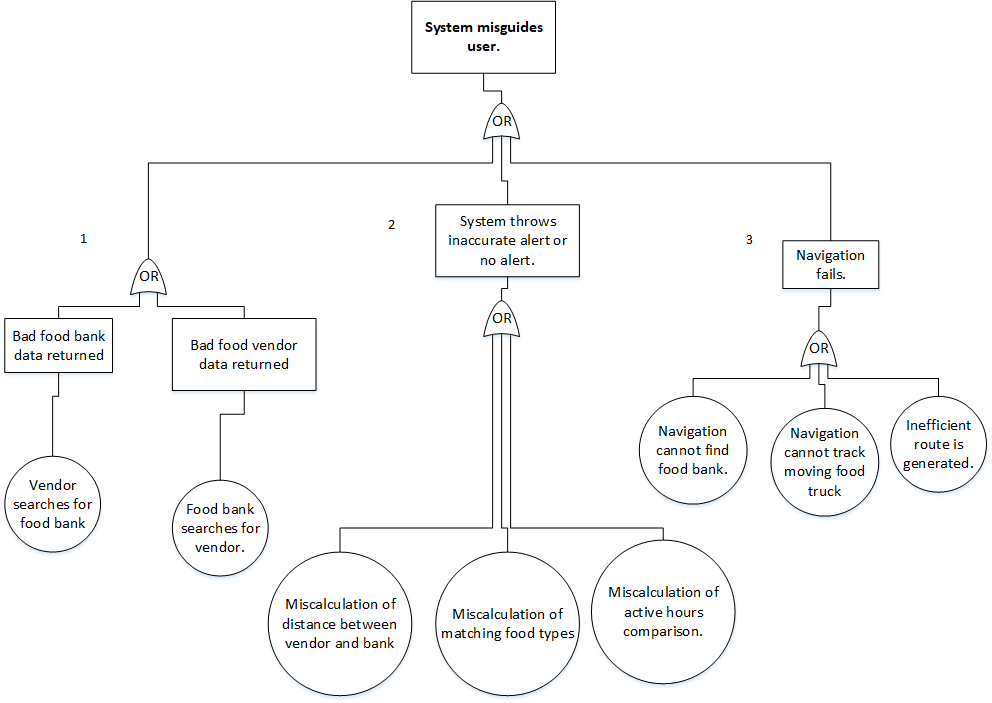


**Explanation of which architecture is more prone to failure:**

Our first high level architecture diagram does not contain a detailed section for registration, however the assumption when it was created was that users would simply register for the system without any sort of account verification taking place. For this reason, this first architecture is much more likely to lead to the fault condition of tax fraud, carried out by illegitimate user accounts. Our second architecture, which includes the App Server sending registration data to the IRS – or some other party that can verify the validity of the user’s business – is explicitly designed to prevent this sort of fraud.

There is, however, nothing in either architecture that would prevent the case of tax fraud being committed by legitimate users who lie to the system, by over or under reporting donation amounts. This may require us to add some sort of additional verification functionality to the system’s donation forms.

Failure Mode 2: System misguides user



**Explanation of which architecture is more prone to failure:**

There is currently nothing in either architecture that completely protects the user from ‘bad data’, which is the root cause of all failures in branches 1 and 2. Our second architecture does, however, list fields that will be expected upon the submission of forms by the user – and these forms are what will populate the system’s database. For this reason, our second architecture would be slightly less likely to return bad data after a user search; however, in reality, a lot more verification would be needed. For example, forms containing locations need to have those locations verified as real places. As another example, food types available for donation from a vendor need to be normalized with food types a bank is accepting, so that the matching algorithm that creates alerts can find a match. Due to the nature of a high level architectural diagram, these types of low level verifications can’t really be expressed; but, they would need to be implemented in order to prevent these types of failures.

Regarding failure due to navigation issues in branch 3, really either architecture is susceptible. Both architectures are reliant upon communication with a 3rd party navigation API, so failure here will be contingent upon the proper functioning of that API. If, however, one architecture were better able to integrate with the API, this would be an advantage.

**Key Attributes**

**Identify the key quality attributes for your system**

Integrity  
This is one of the most important if not the most important attribute for this system. During registration, the system will be collecting sensitive credentials from the user, such as tax payer ids. The system needs to store this information securely. Additionally, the system will be collecting donation records that can be used for tax reporting. These records need to be: accurate, securely transferred and stored so that only the user they are associated to can access them (i.e., encrypted), and stored in a way such that they cannot be modified after they are created.

Efficiency  
This attribute will be crucial to a functional system. Essentially all of the system features accessible to users via the UI can modify the system’s database - and at any given time a large number of users could be using these features! In addition, the state of the database can affect other system features, such as automated alert generation and the downloading of donation records. For these reasons, it is crucial that the system’s database can perform a large number of queries efficiently, and that the system’s UI can respond accordingly.

Reliability  
Once a user base has been cultivated, system reliability will be important to keeping them around. At its core, the system will be used to help users schedule their work day. For vendors, this means creating an efficient route to recipient food banks; for food banks, this means scheduling drop offs from vendors. If the system is not reliable, these users will find another method of scheduling.

Usability  
This attribute applies to any software system, but is especially pertinent to this system. With a bit of work, a person can already do most of the things this system enables via Google, a few phone calls, and a bit of responsibility when it comes to maintaining their tax records. To be successful, using this system needs to be easier than managing food donation via disparate processes.

Flexibility  
There are a lot of different food vendors serving a lot of different foods. The system’s database needs to be able to store potentially any type of food a vendor wants to donate and match that food with the needs of local food banks.

**Assess how well each of the two architectures supports each quality attribute**

Architecture #1

Integrity  
This architecture does include user authentication within the server, but it does not explicitly state that it will include encryption during transmission. The details of this need to be fleshed out at a lower level. Another factor that could compromise the integrity of this architecture is that it communicates with an external, 3rd party resource to calculate navigation. We would need to ensure that this connection is secure and that it does not allow the 3rd party to access sensitive user information.

Efficiency  
This architecture performs a number of services within its app server, however it offloads location calculation to an external resource. This would definitely increase efficiency. This architecture also utilizes Angular, which, like many Javascript based libraries can perform processing in the front end, decreasing the load on the server.

Reliability  
There is nothing to indicate that this architecture won’t be reliable. It does, however, only reference a single app server. If this single app server goes down, the system will become inaccessible. To avoid this situation, a cluster of app servers could be used.

Usability  
This architecture utilizes node.js, Express, and Angular. These Javascript based technologies are ideal for creating a user centric experience on the web.

Flexibility

This architecture utilizes a NoSQL database. This type of database could potentially be more flexible than a traditional, relational database.

Architecture #2

Integrity  
This architecture does include identity verification, however to increase integrity the addition of an encryption service that is called during data transmission and data storage in the database is needed.

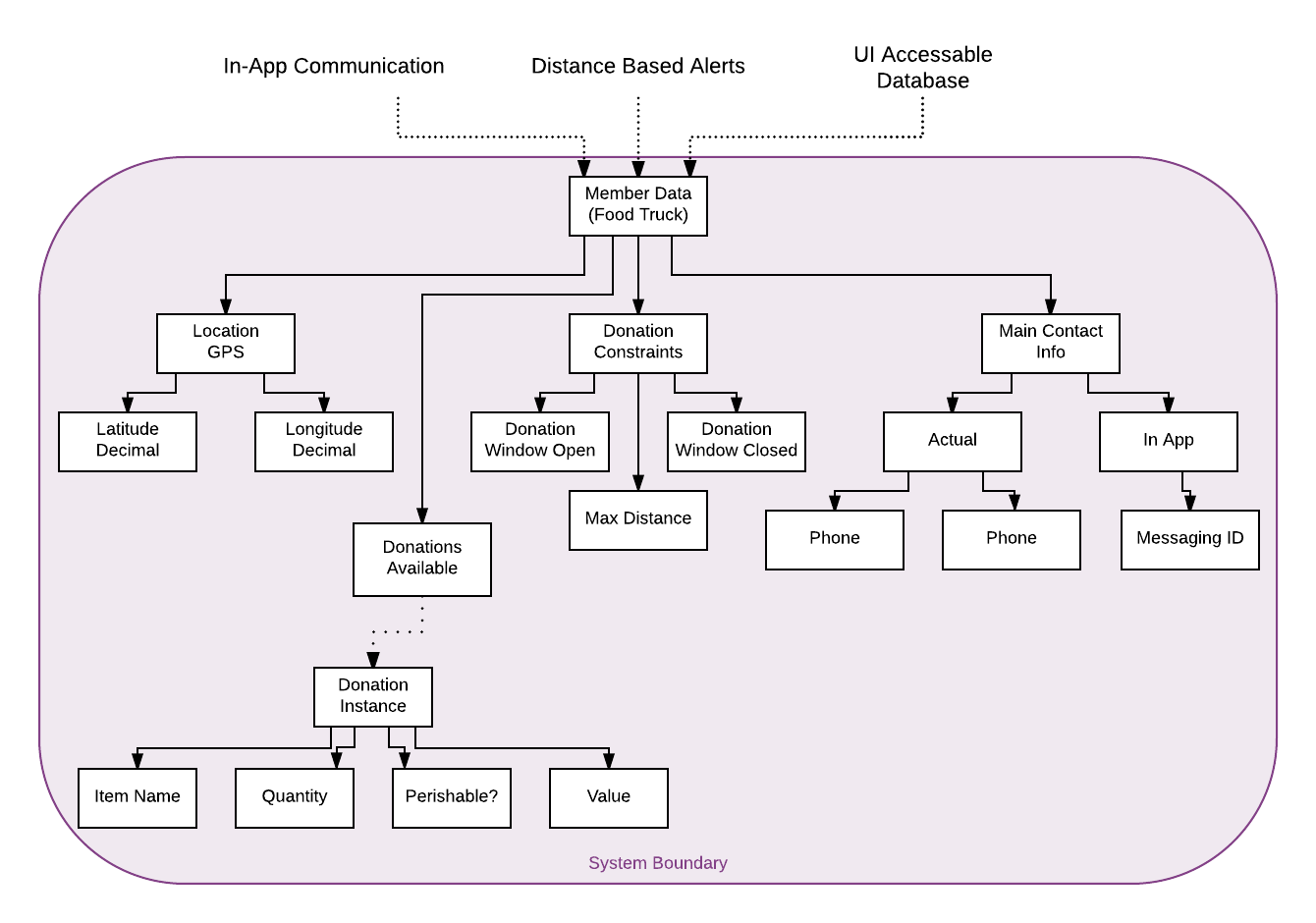
Efficiency  
This architecture has the app server performing a number of services, which may need to be provided simultaneously. While this may not be an issue, we could potentially off load some of the services, such as location calculation, to an external resource. Offloading these computations could increase efficiency.

Reliability  
There is nothing to indicate that this architecture won’t be reliable. It does, however, only reference a single app server. If this single app server goes down, the system will become inaccessible. To avoid this situation, a cluster of app servers could be used.

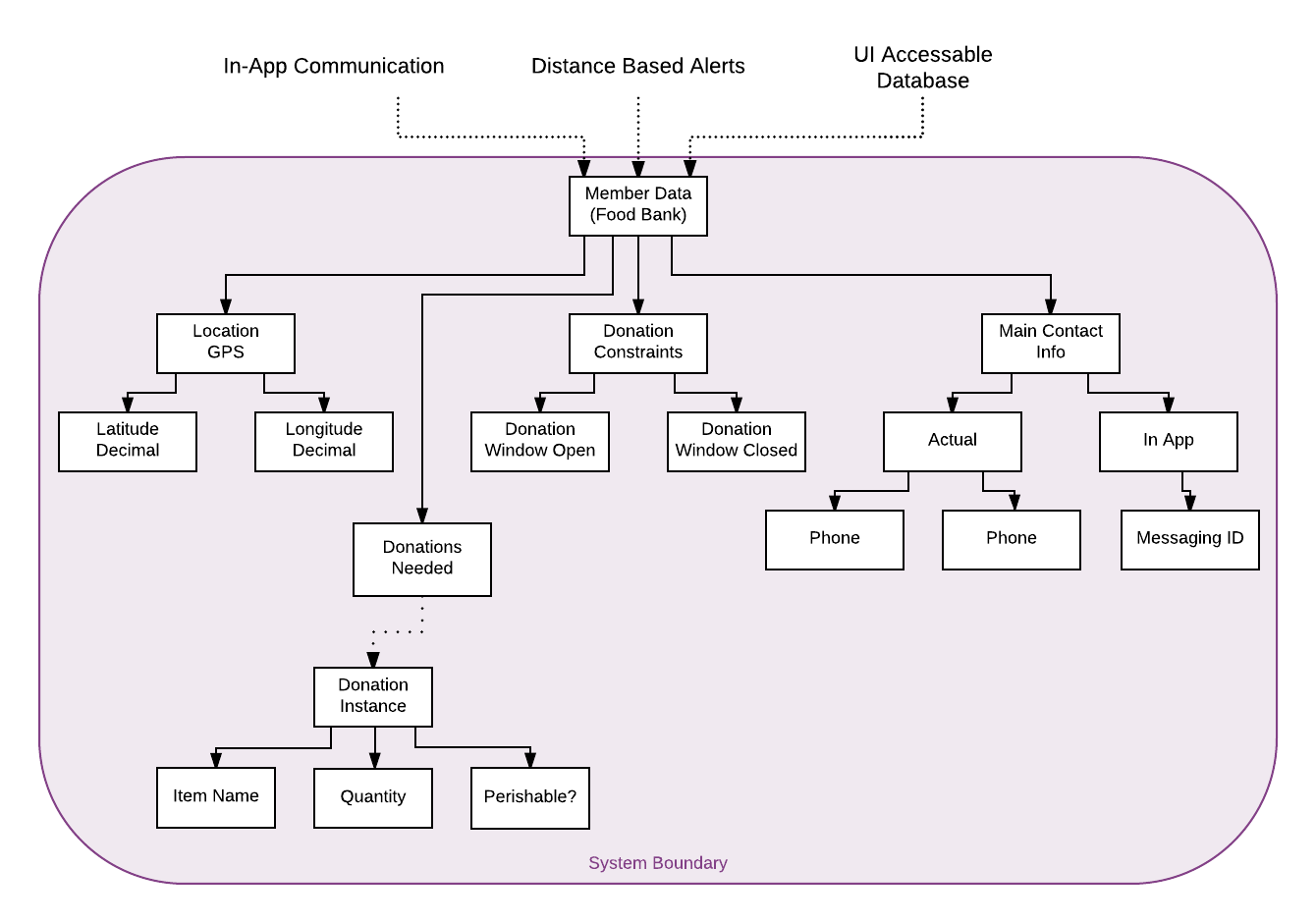
Usability  
This architecture has a design where the user simply needs to fill out forms within the system’s UI. After the forms are filled out, the app server essentially takes care of the rest. This type of design has the potential to be very user friendly.

Flexibility  
Assuming the client side forms allow the user to enter whichever values they like for food types, and assuming the app server can normalize those values before they are stored in the database, this architecture should be flexible enough for the system’s needs.

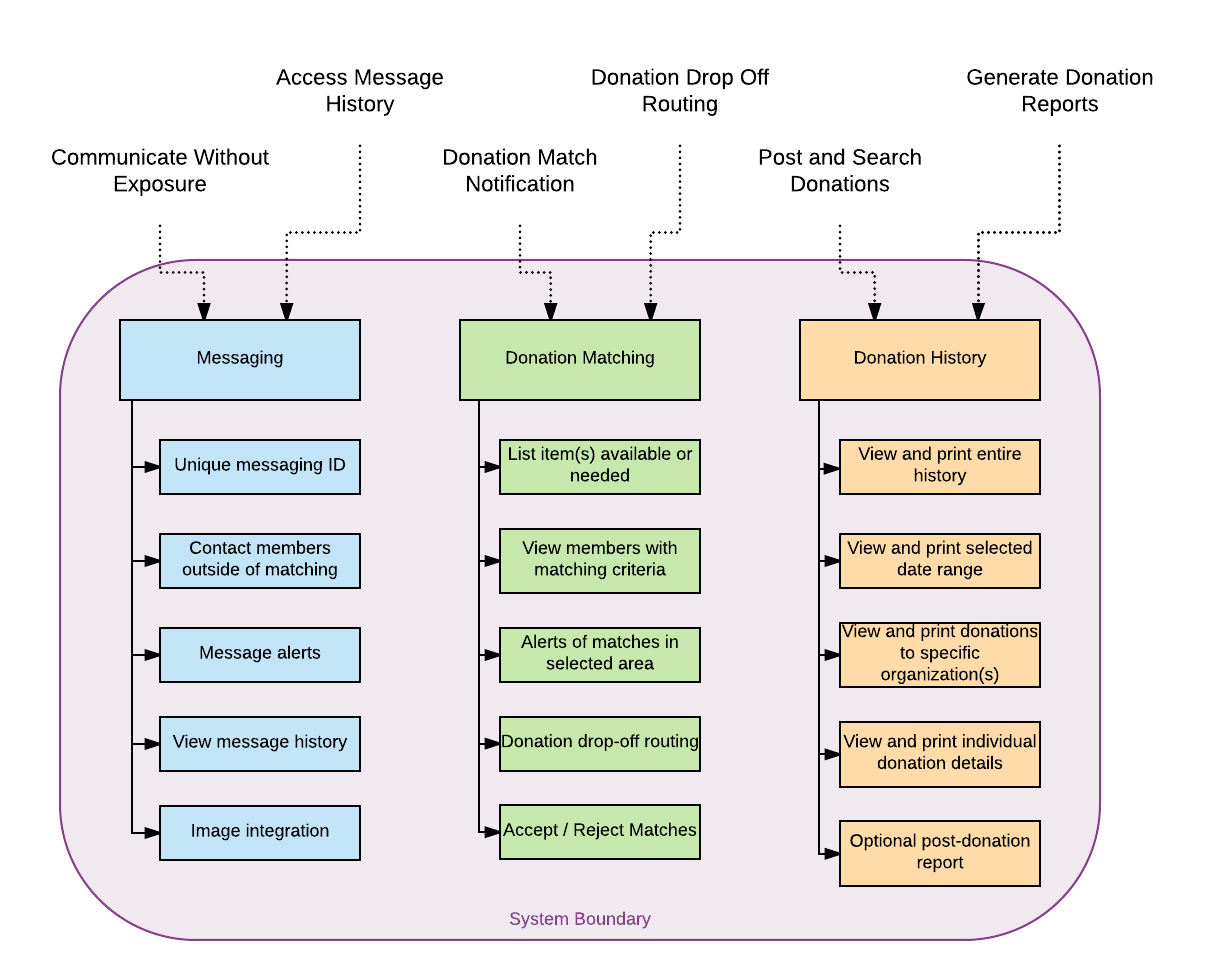
#### Further decomposition

Data-Oriented Decomposition of Database: Food Truck

Data-Oriented Decomposition of Database: Food Bank



Feature-Oriented Decomposition of Client



#### Validate Architecture By Use Case

For walking through the selected use cases, we have decided to use the second high-level architecture. This architecture tends to be more complete as it relates to use cases.

Use case 1: Vendors finding food banks

Pre-conditions:

* The diagram satisfies the requirement that the vendor and the food banks be registered in the server.
* Any food bank users can post food needed, and the results can be traced by the server and be shown to any food truck owner.
* The diagram meets the requirement to check if the food truck is within maximum distance of a food bank, to check if the food bank is within open hours, and to check if the food type is satisfied by food vendor.

Post-conditions:

* The diagram allows food vendor to complete a food donation online.

Event of flow:

* The diagram allows vendor to post a donation request. Vendor can use images and choose food type categories to describe. The post data will be stored in the database.
* The diagram allows the server to push a notification to food banks when they find matching results.
* The diagram meets the requirement to build a connection between vendor and food bank when the result is accepted by vendors.
* The diagram can calculate the most efficient route. But it needs improvement, since the diagram does not specify another maps API or in-app calculation.
* The diagram can record a complete donation in the system. But to improve, the diagram does not specify to deactivate the original post once the donation is completed.

Use case 2: Food bank finds donation

Preconditions:

* The diagram allows food banks to register in the server.

Post conditions:.

* The diagram allows food banks to search vendors, to make contact with the vendor in the APP.

Event of flow:

* The diagram allows food bank to post food needed by filling a donation request.
* The diagram allows the donation request to be stored and be searchable and be shown to any vendors in the app.
* The diagram meets the requirement to check if the food truck is within maximum distance to a food bank, to check if the food bank is within open hours, and to check if the food type is satisfied by food vendor.
* The diagram allows vendor to send feedback on any post to the food bank.
* The diagram allows food banks to respond to feedback and set up a working connection in the APP.

Use case 3: Communication between vendor and bank

Preconditions:

* The diagram allows both food bank and vendor to register in the APP.

Postconditions:

* The diagram allows food bank or vendor to internally contact the server about a potential donation.

Event of flow:

* The diagram allows food bank or vendor to fill a donation form and save the data in the server.
* The diagram allows food bank and vendor to send and receive message directly. Then the potential donation can push an acceptance or rejection message.
* The diagram allows a donation scheduling message to be push from one party to another.
* The diagram allows once a donation is delivered, the donation will be marked as complete and been stored in donation history.

Use case 4: Generating Donation Records

Preconditions:

* The diagram allows vendor to register in the APP.

Postconditions:

* The diagram allows vendor to pull out donation history, search for a donation details, and print out the receipt.

Event of flow:

* The diagram allows vendor to pull out either full history or partial history filtered by date.
* The diagram allows vendor to view and print all donation logs.
* The diagram allows vendor to view a detailed page on a selected donation.
* The diagram allows vendor to print out a detailed page.

#### Explain the implications

After tracing the use cases through the architecture, it has been determined that certain portions of the architecture need to be fleshed out some and added to. The areas that need to be expanded upon are the donation architecture flow diagram and the tax receipt architecture flow diagram.

The donation architecture diagram needs to be improved by having one of its subprocesses fleshed out into its own diagram. This portion shows the app server searching for nearby clients and calculating the shortest route to a food bank. These two processes are large in and of themselves. They could easily be broken down into their own architectures involving how the routes are calculated, what objects are used in the calculations, how the nearby clients are found, and whether or not those calculations are offloaded to an external API. This change would enhance the efficiency key attribute as well as reduce the potential for failure.

The efficiency attribute would be enhanced as with a subprocess having its own architecture, that architecture could be traced for redundancies or inefficiencies. If we see a process being done twice, the entire architecture of the system could be redesigned as needed to eliminate the redundancy. If the flow of the subprocess becomes too complex this may be an indicator not only of application inefficiency but also that it might be an inefficient use of team time to build it out. Therefore, the decision could be made to look for tools or services already in existence to handle the task for us. A decision such as that could additionally increase reliability, as an outside service may have more experience and expertise than the team on hand in designing and implementing that portion. The potential for failure would be reduced as by analyzing the architecture of a subprocess, it could be traced for failure points. Once these failure points are identified, corrective measures could be applied to either the architecture of the subprocess itself, or to the entire high level architecture.

The tax receipt portion of the architecture needs to be re-analyzed and possibly redefined from the perspective of one of our failure points. We found that it may be too easy for a food bank and vendor to work together to forge donations. Thus, as a group we must brainstorm some sort of process to prevent this. For example, maybe there is a way to do an automated audit regarding pictures from vendors and banks or a tracking system that follows the truck to the bank. Either way, the architecture will have to be to modified to incorporate this.

This would significantly increase the integrity quality of our application. The fact that our application can be used to generate tax receipts that will be submitted to the IRS requires the utmost integrity on the part of our system. The information we produce must be accurate and should be a tool for preventing criminal activity. Therefore, by architecting a solution to this, we will be fulfilling the requirement that our application maintain its activity. Additionally, this eliminates one of the failure points we identified, and thus by definition, reduces the potential for failure.

Overall, our architecture fulfilled the needs of our use cases and of our system fairly well. The process of generating the architecture and tracing the use cases through it allowed us to see where we had not thought a process all the way through or did not fully design a portion of our system. By tweaking the architecture to address these points, the points of failure are reduced and a chain effect is created whereby improving one key attribute others can be improved as well.