Process:

Risk Assessment
The top risks to the successful completion of our project are related to the short amount of time we have to complete it and the lack of experience of our team.
Risk 1: It will be too difficult to write an app that works on both iOS and Android in the remainder of the quarter.

Likelihood of occurring: high
Impact if it occurs: medium
Evidence: Nobody on our team has used React Native, the service used to port to iOS and Android. Many people on the team have no experience with JavaScript, the language used in React Native. Many people on the team don't have access to an Apple computer, which is required for iOS development.
Steps to reduce risk: We are actively practicing with JavaScript and React Native.
Plan for detection: We will continue to make well-thought-out schedules and frequently evaluate our progress.
Mitigation plan: Only write the Android app.

Risk 2: Some more complex features will be too time-consuming or complicated to implement on a tight schedule, namely an effective search algorithm and a map display of search results.

Likelihood of occurring: high
Impact if it occurs: medium
Evidence: At least one member of the team has worked on search features in other apps, and worked in teams that implemented a similar map feature, and from those experiences knows that these features are very time-consuming to implement on even one platform, even with experienced developers working 40 hours a week. We are trying to implement them on two platforms while working substantially fewer hours per week.
Steps to reduce risk: We will start on these features early, but also make sure not to let work on them impede the completion of simpler or more important features.
Plan for detection: We will continue to make well-thought-out schedules and frequently evaluate our progress.
Mitigation plan: Only display the search results as a list, not a map.

Risk 3: We will have difficulty integrating the database with the front-end.

Likelihood of occurring: low
Impact if it occurs: high
Evidence: Most members of the team have experience with frontend or backend, but no members have experience with both.
Steps to reduce risk: The database is fairly simple and it should be easy to access, so this risk isn't too great. The main concern would be that we have difficulty determining what to store in the database and how to format/access it efficiently, so we will focus on getting this right from the start.

Plan for detection: We will know if this is a problem because the app won't function unless we are able to connect the two components.
Mitigation plan: There really isn't one. We have to get this right. Accordingly, we have organized our timeline to complete the implementation of database features early in development to ensure we can accomplish them.

Risk 4: The team's overall lack of experience with mobile development - and complete lack of experience with React Native - will make it take longer than we estimated to complete each component, either because we spend too much time researching or don't get the architecture/style right on the first attempt, leading to bugs and performance/scalability issues.

Likelihood of occurring: high
Impact if it occurs: high
Evidence: No members of the team have experience with React Native. We have not completed any features yet so we don't have any evidence to base our scheduling off of. The majority of members of the team have no experience with mobile development or the design patterns involved in it.
Steps to reduce risk: We will start on the initial features ASAP to get a better idea of how long each one will take, and make adjustments accordingly. We will set up continuous integration on GitHub to ensure that architecture changes/bug fixes that we implement as we gain experience don't break pre-existing code.
Plan for detection: After completing the initial features, we should have a better idea of how long the remaining ones will take.
Mitigation plan: We will cut or simplify features if we can't finish all of them on time.

Risk 5: The app will have security/authentication issues.

Likelihood of occurring: high
Impact if it occurs: low (in terms of the class), high (in terms of the real-world usability of the application)
Evidence: We have made some design decisions in the login process that may make it difficult to authenticate organizations or users. Most members of the team have no experience with computer security.
Steps to reduce risk: If time allows, we will research more effective authentication techniques for mobile applications and attempt to implement one.
Plan for detection: If time allows, we will design tests for authentication and other potential security issues.
Mitigation plan: Don't include the optimal security/authentication features in the version of the app completed by the end of this class if they are too time-consuming to implement.

Project Schedule

Timeline:

Task	**Person**	**Effort Estimate**	**Dependencies**	**Due Date**
Completed Skeleton				
Set-up tools on personal machine	All	3 hours	None	10/25
Create blank pages for all prospective screen	1 dev	4 Days	Set-up tools on personal machine	10/25
Database Plumbing				
API and authenticated endpoint	2 devs	4 Days	Database setup	10/25
3 instances of dummy data in tables	2 devs	3 Days	Database setup	10/28
Tech Ramp-Up				
Javascript and Python practice	All	7 Days	None	10/28
Request Functionality - Basic functionality complete after this				
Organization can make requests	4 devs	7 Days	Basic screens; API/authenticated endpoint	11/4
Requests displayed in feed	3 devs	7 Days	Basic screens; API/authenticated endpoint	11/11
Search requests	4 devs	7 Days	Basic screens; API/authenticated endpoint	11/11
Edit, Remove requests	2 devs	5 days	Requests displayed in feed	11/18
Requests displayed in map	4 devs	7 days	Basic screens; API/authenticated endpoint	11/18
Subscriptions - Extended functionality				
View organization pop-up	3 devs	3 days	Basic screens	
Subscribe to organizations	3 devs	4 days	View organization pop-up; API/authenticated endpoint	11/25

Log-in				
Log-in as an organization or as a donor	4 devs	7 days	Basic screens	11/25
Release Candidate				
Bug fixes, polished product, fully functional and reliable	7 devs	7 days	All previous tasks	12/02

Team Structure
The majority of the HandOff team members have a high interest in taking part in the coding portion of the application. Kennan and Erin will be the heads of front-end development, as they both have the most experience with front-end work while Seth and Brendan will be the heads of back-end development. Sam, Ian, and Peter will be taking a station of flexibility, and will separately move between back-end and front-end development depending on team needs and overall progress. Python and Javascript will both be heavily used in application development, so all team members who are not familiar with either language are responsible for teaching themselves and getting used to coding in both. Brendan is responsible for setting up and redeeming AWS while Erin will be responsible for pioneering the UI and planning its general direction with the other front-end coders. As our application grows and we have a larger amount of code, testing will be required by all members. Everyone must write thorough tests for the code that they contribute. To ensure high quality tests are written, Kennan will review tests written by others on a weekly or semi-weekly basis.
Team communication will be primarily done during twice weekly meetings on Tuesdays and on Thursdays from 9:30 – 10:20 at the reserved Odegaard room and online through a team Slack messaging board. The brunt of the brainstorming, application development ideas, and planning will be done on the team Google-doc, which will be edited during twice weekly meetings. Members of the team who are involved in the same or similar areas of the application will likely meet in person to work together and will coordinate online as well.

Test Plan

Our testing strategy is comprised of unit tests, system tests, usability tests, and UI tests. In addition, we will do manual user testing on the app. This testing strategy should be adequate as long as it is followed; aside from some leniency with code coverage in unit tests, the testing strategy is very thorough and most tests are performed with continuous integration.

We will use GitHub issues to track issues that we find and ensure that they are addressed in a timely manner. In addition, we have Slack to discuss issues, so any questions that arise regarding a particular GitHub issue can be answered and discussed on our channel.

Unit Test Strategy:

-Each class and each top-level function should have a set of unit tests associated with it.

-Whoever writes the class/function will also write the test for it.

-Tests will be automated with continuous integration. We will most likely use Travis for continuous integration.

-Each function test should have 100% code coverage if it's practical. Otherwise, a few general cases should be tested as well as as many edge cases are practical.

System Test Strategy:

-Each endpoint in the database the mobile code calls should have an associated System Test.

-The tests will be developed on the mobile side.

-The tests should call the endpoint with various possible parameters.

-These will be run manually every time there is a significant update to the database.

Usability Test Strategy:

-Our main usability testing will be manual, i.e. we will give prototypes to people who haven't seen the app and see if they can figure out how to use it. We will do this for each new feature/major update.

UI Testing

-In addition, we will have automated UI tests for the mobile side, for each screen. These will just test that the screen appears as expected and the buttons, navigation, etc. behave as expected. These will run with continuous integration.

Documentation Plan

Our documentation plan will be pretty simple to ensure that those who read it may quickly understand it. However, the documentation will be sufficient to ensure that our application is understandable, maintainable, and easily usable.

For Users

We will have in-app documentation that tells users how to use the app. We will write this towards the end of the quarter, after we have a better idea of exactly what features will make it into the app, and have tested prototypes with potential users to get a better idea of what is unclear and what requires instruction. If time allows, this documentation can involve interactive pop-ups, but may be as simple as a help page.

For Developers

The general rule for developers is that anything unclear should be documented:

-Most variables, functions, and classes in the code should have at least one sentence of documentation.

-Any process involving multiple steps that needs to be repeated (e.g. generating a code signing identity in Xcode or publishing to the App Store/Play Store) should be documented using a page in the GitHub Wiki for the project.

Coding style guidelines

Java:

https://source.android.com/source/code-style.html (android-specific, takes precedence over latter for front-end)

https://google.github.io/styleguide/javaguide.html

Swift:

https://github.com/raywenderlich/swift-style-guide

JavaScript:

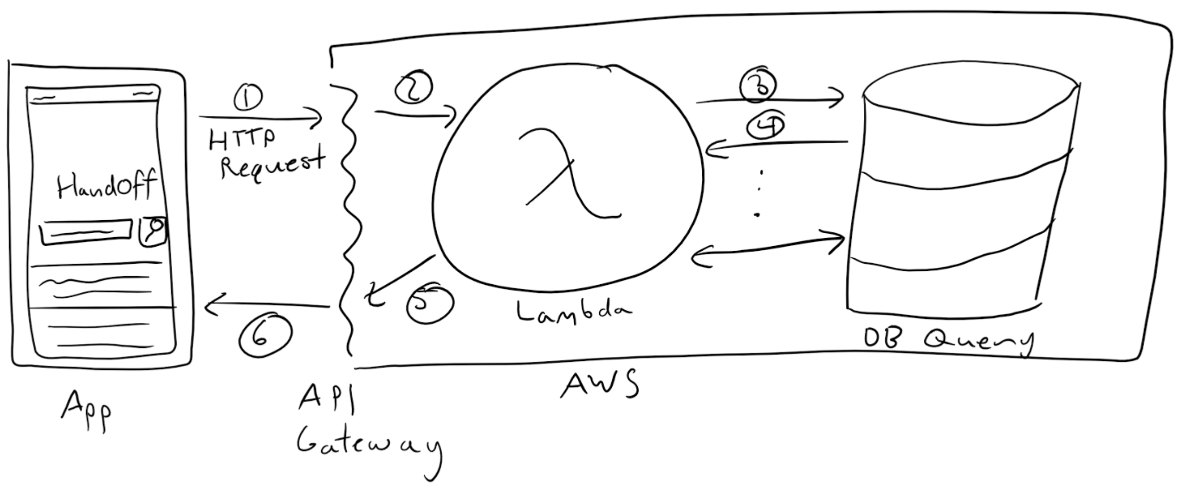
https://google.github.io/styleguide/javascriptguide.xml

The coding style guidelines will be loosely enforced since it is difficult to write automated tests that ensure they are followed. It will just be expected that all group members read the style guidelines and make an effort to follow them as they write code. Additionally, any peer-reviews we do of code should include a check for style. As a

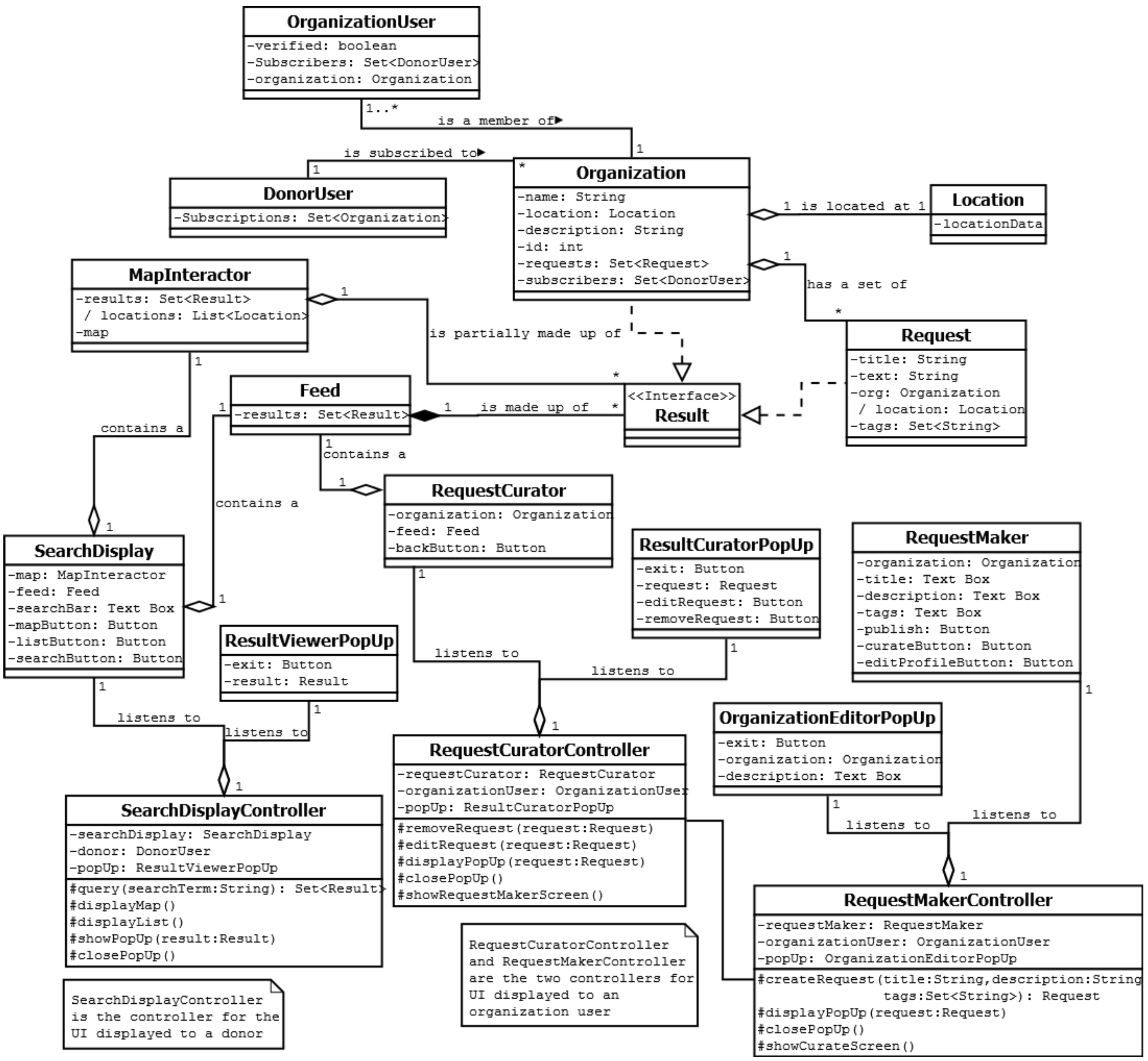
Design:

System Architecture

Users will interact with HandOff through mobile applications. These applications will make HTTP requests to an Amazon API Gateway, which will trigger the appropriate Amazon Lambda functions. These functions will modify an Amazon DynamoDB table accordingly. API Gateway will provide generated SDKs that the mobile applications can use to make API calls. API Gateway will be configured to trigger the execution of an appropriate Amazon Lambda function upon receipt of a request. This function will implement the backend logic required to carry out the request, which will involve reading and writing to DynamoDB, then returning a request to the client via the API Gateway. A graphic of this interaction is shown below.


HandOff
1
HTTP Request
2
3
4
5
6
Lambda
DB Query
AWS
App
API Gateway

UML Class Diagrams



RequestCuratorController and RequestMakerController are the two controllers for UI displayed to an organization user

SearchDisplayController is the controller for the UI displayed to a donor

The database on AWS will store three tables: organization, requests, and users. The schema of these tables is listed below.

Schema

Organization

- UUID (PK)
- Name
- Location
- Description
- Request Set
- Subscriber set

Requests

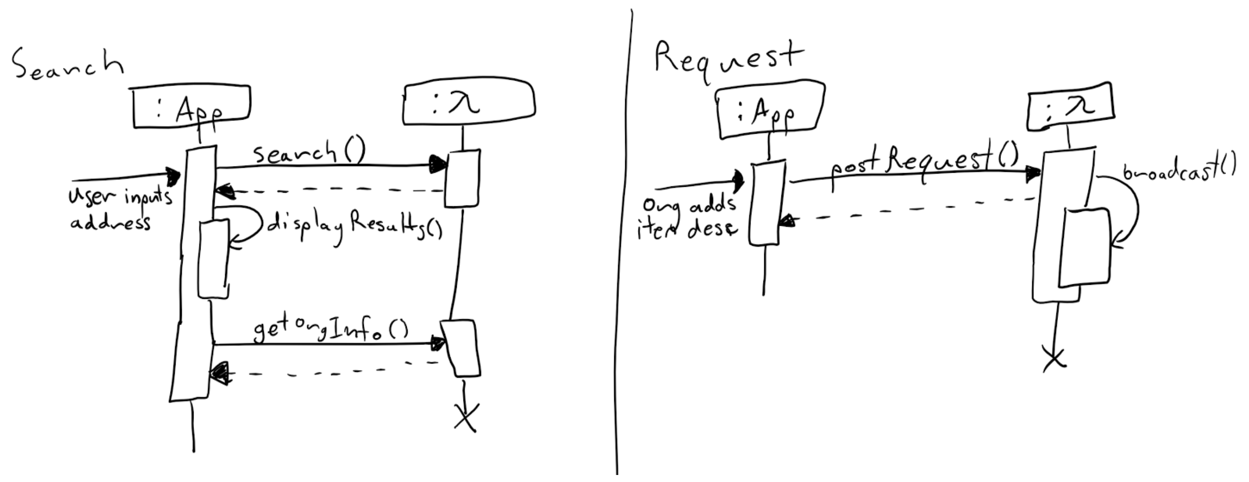
- UUID (PK)
- Org UUID
- Title
- Text
- Tag set

Users

- UUID (PK)
- Organization set

UML Sequence Diagrams

Shown below are two UML diagrams. The left diagram represents a donor searching for organizations around a certain address. The right diagram represents an organization posting a request. The lambda sign represents the Amazon Lambda service we will be using to manager servers.



Alternative architectures

#1

We considered Amazon EC2 instances with web servers to handle API requests. These web servers would then make requests to DynamoDB. Then we realized web servers would have a much higher operational cost for the team (e.g.: we'd have to manage security upgrades). So instead we replaced them with API Gateway and Lambda.

#2
We also considered DynamoDB as a database service. However, we were discouraged by the high pricing. A modest number of provisioned reads and writes would constitute the vast majority of our recurring costs. We considered using traditional SQL databases running on EC2 instances, but then decided that the operational costs would be too high. We also opted for NoSQL because it allows a flexible schema, which is important as our application is highly likely to change during development.

#3
Initially, we looked into using Xamarin or Apache Cordova for our front-end, but we instead chose React Native. We decided against Xamarin because few of our group members have used C#, and it seemed to be more complex to setup than React. Cordova was eliminated pretty early, as it is similar to React Native but less well documented and supported. Ultimately, we settled on React Native because of its extensive documentation, popularity, HTML/CSS/JS code and ability to transition to React.js if we wanted a website.