Becky Leslie (beleslie)
Bryce Martz (bmartz)
Daniel Houtsma (dhoutsma)
David Li (lidav6)
Dylan Whitlow (whitldy)
Nick Bissiri (nbissir)

System Architecture

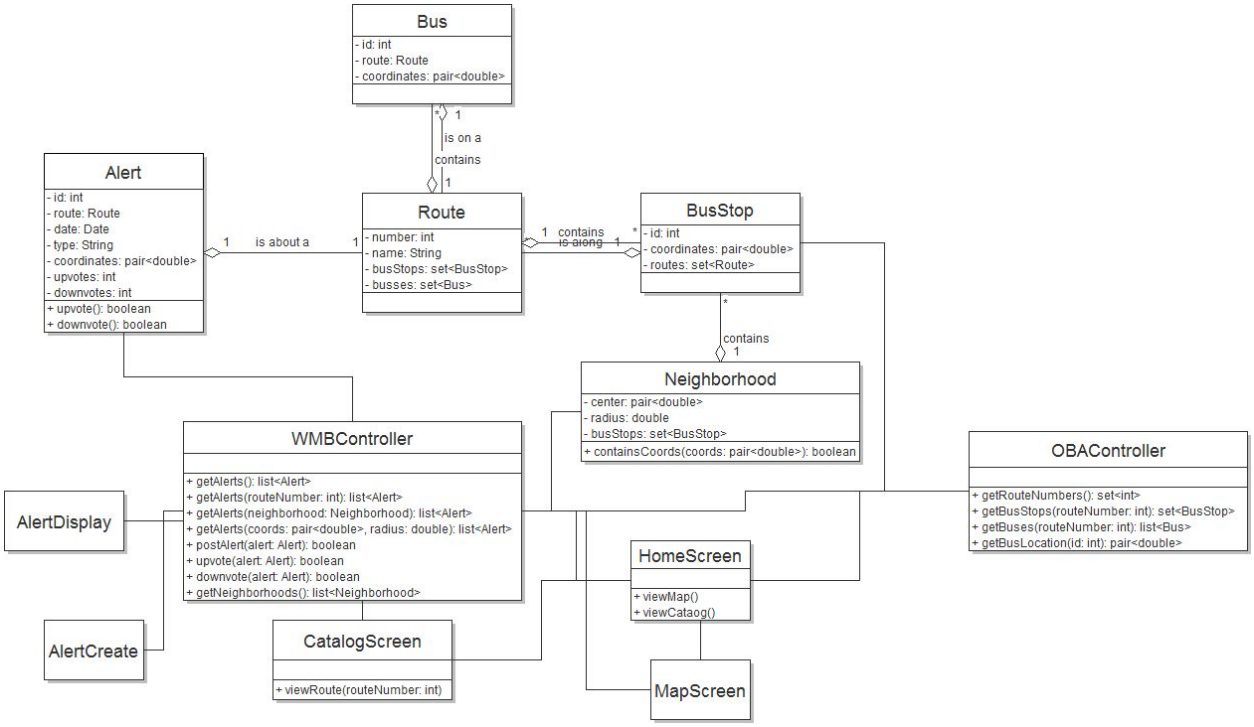
At the highest level the modules are: the database storing user-submitted alerts, the Rails server fetching information from the database and OneBusAway's public API, and the Android user application.

The user app will submit requests to the server over HTTP, providing url parameters parsed by the Rails server. For example, the user app will request bus route data to display a list of routes to the user. The server will respond to HTTP requests with JSON generated by Rails from the relevant objects. The user app will parse the JSON and build its own objects. Alerts will be formatted in the database as generated automatically in Rails from alert objects.

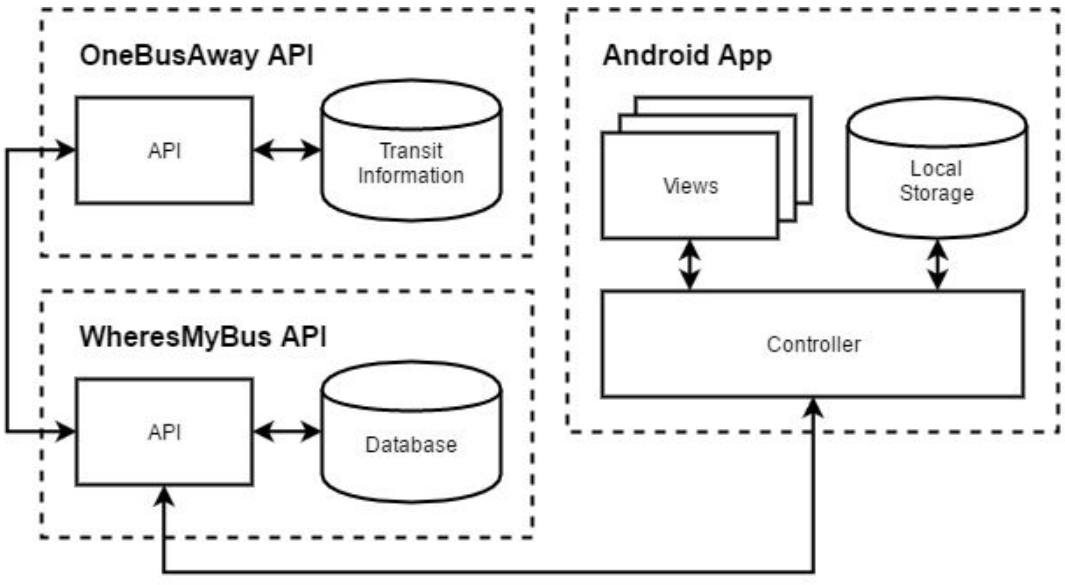
For non-alerts, almost all data will come from OneBusAway's public API, which will be accessed for any request from a front end user. We considered a design in which the front end application would directly request this information from OneBusAway's API, but decided that it would be simpler for our front end to only interact with our back end. We also considered a design in which data from OneBusAway would be stored by the server and periodically refreshed. We felt that we would want to update this data so frequently that storing it all would be redundant.

There are several kinds of data: alerts, busses, bus routes, bus stops, and neighborhoods. Each bus route contains multiple bus stops, and each bus corresponds to some route. Busses and stops have location coordinates used for display on a map. Neighborhoods are regions of 2D space used in the user app select a subset of all alerts, routes, stops and busses for display. Alerts contain coordinates, text, date/time, duration, route, score, and bus ID. We are assuming that hard-coding neighborhood regions is a feasible task to complete, at-least for King County (for regions outside of Seattle we may choose to include only cities as 'neighborhoods'), but we considered replacing neighborhood features with user defined regions in the front end, such as with 'dropped pins' on a map.

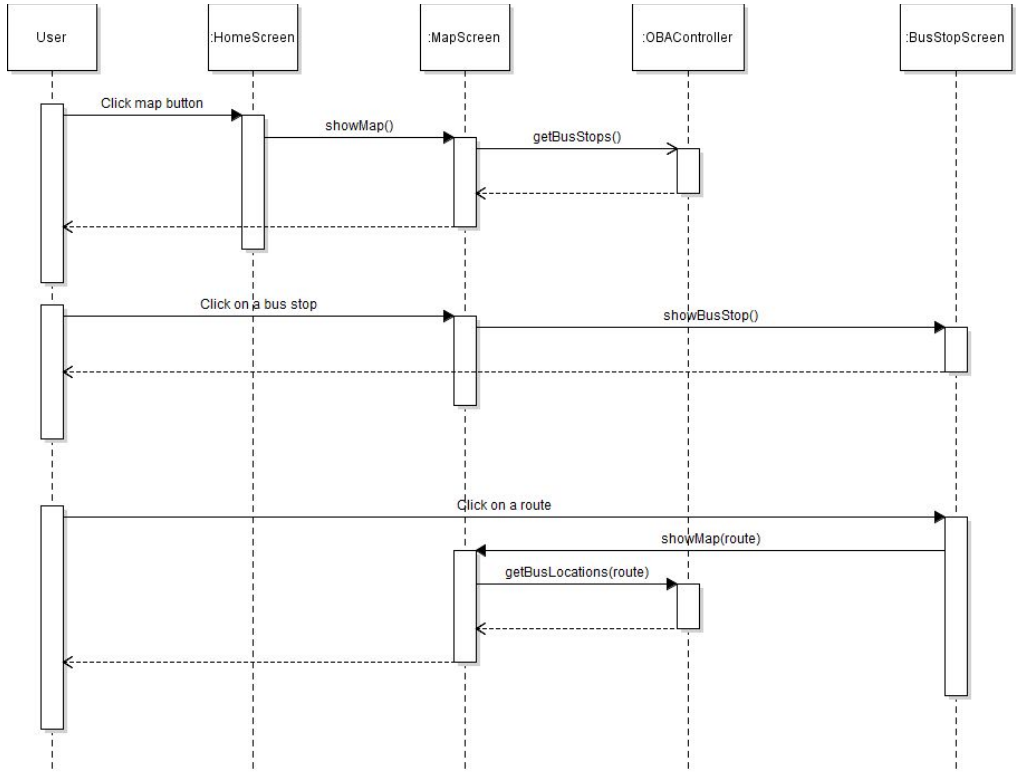
What follows are several diagrams demonstrating the system architecture.



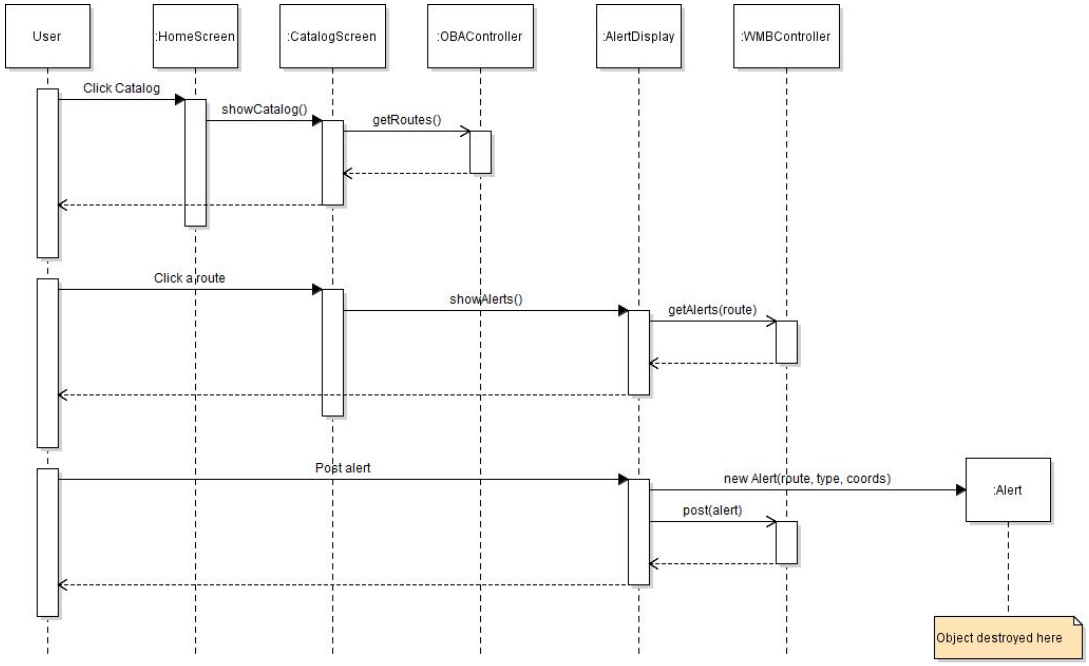
UML Class Diagram



System Architecture Diagram



UML Sequence Diagram - viewing a bus on the map



UML Sequence Diagram - posting an alert

Process

Risk Assessment

One of the main risks that may prevent us from building a functional app is being able to test it in the real world. Since we are including all the bus routes in King County, our app has a fairly large scope. This will make it difficult to simulate how it will actually be used by real people. It seems fairly likely that we will encounter this risk in our project. Fortunately though, it's impact will be relatively low because as long as the rest of our system is tested and working, there probably won't be that many scenarios that require large scale, real life testing. We have come to these conclusions based on the fact that there are dozens of bus routes in Seattle, each with multiple buses and a large amount of bus stops. So In order to address this problem, we figured we could either recruit a lot of people to actually go out and use our app, or just simulate the data instead.

In addition to providing a forum for alerts about specific bus routes, we wanted to implement the same kind of forum for neighborhood alerts, where people could submit information about situations like construction that would affect all the routes in part of a given neighborhood. This feature poses the risk, though, of distinguishing neighborhoods. The location data to distinguish the perimeter of every neighborhood, even in Seattle alone, would consume a large amount of storage. Furthermore, outside of the city, the distinction between different neighborhoods within a town might be more ambiguous. Since neighborhoods are defined by uneven borders, the risk of the location data for each perimeter requiring a lot of storage space has a high likelihood of occurring. If our servers cannot store the data for each neighborhood in King County or even Seattle, the risk would have a medium impact on our project because we would not be able to implement the forum for neighborhood alerts, one of our main features described in the SRS. Seattle alone has 127 neighborhoods, each of which likely have a pretty complex boundary that would require a lot of data to encode. To reduce the impact of this risk, we have been researching ways to get this information from a different source so we don't have to store it ourselves. We can detect the problem by attempting to store the data ourselves and see if it is too large for our database. If this risk does end up happening, we will instead implement a feature to find alerts near the user. Although it wouldn't be as granular, it would still provide the user with similar functionality.

As one of our stretch features, we wanted to parse information provided by King County Metro on their website and through their email and text alerts to post the information for users of our app. The major risk of this feature is that King County Metro might not submit alerts in a consistent format, which would prevent us from building a parser that could find the information we want to post. The likelihood of this happening would probably be medium to low and if it did, it would have a high impact. That is because if the format changes, our parser would be extracting gibberish which when posted, would be confusing to the users. Our estimate is based on us looking at a lot of the alert files King County Metro posts on their website and seeing that they all follow a pretty similar format. To reduce the likelihood of the risk happening, we could write a parsing test that verifies the format of an alert file, which is a PDF. We could then use the test to periodically check if all files currently posted on the King County Metro alerts page pass. For now, our mitigation plan would be to just fix the

parser to work with whatever different format they change to. We could then try to work with them to see if they would be willing to submit the alerts in our app directly.

Another risk we could potentially face is not finding out the optimal time for destroying our alerts. On one hand, if we keep the data for too long, it could be confusing to the user as they will see alerts for issues that have already been resolved. On the other hand, if we destroy it too early, we prevent the user from getting the information they're looking for. The chances of this problem occurring are relatively high. Different types of alerts will likely need to be active for different amounts of time and so it's hard to determine the optimal duration for all of them. This would have a medium impact as it could inconvenience the user a little bit, but it won't prevent the core features from working. Our estimates are based on the discussions we've had about the various types of issues that can affect busses and how long they would typically last. To reduce the likelihood of this risk, we'll do some research on the web for common problems that impact busses and how long they would be affected. We would also probably err on the side of caution and keep the alerts around for a little longer than we think because having the alert there and not needing it is better than needing an alert and it not being there. To see if this is actually a problem, we can check if users are complaining about whether alerts are lasting too long or not. For our mitigation plan, we could change our implementation to allow users to manually resolve alerts if they think the issue isn't applicable any more.

One last risk would be not getting enough alerts from the users. Our service will rely extensively on crowd sourced alerts and if people aren’t posting enough of them, others won’t want to use our app because of that. This risk probably has a higher chance of occurring because we don’t have a budget for marketing and we will likely rely on word of mouth to get people to use it. And if this risk were to happen, it would have a very high impact, because without any users posting alerts, our app won’t be as useful as we intended. Our estimates for this risk come from the realization that we need a decent amount of people using the app to make it fully functional. To reduce both the likelihood and the impact of this risk, we will utilize other sources of alerts, from transit agencies for instance, so we are still able to provide value to our users. We can detect if there are any problems by monitoring usage statistics such as number of active users or alert postings. In the case where this risk does occur, we can attempt to market it on social media and other platforms in order to get some traction and then hopefully there will be enough people submitting alerts to draws other people in too.

Project Schedule

Task #	Major Milestone	Tasks to Achieve the Milestone	Length of Effort	Depends on Task #
1	Software Requirements Specification (due October 14)	a) Identify the major features of our app b) Design UI diagrams c) Develop use cases d) Decide what software tools we will use e) Draft a schedule for the project	1 week	n/a

		f) Pinpoint the major risks g) Create website, GitHub repo, and mailing list		
2	Software Design Specification (due October 21)	a) Define the system architecture b) Assess and develop plans to mitigate the major risks c) Determine a plan for testing d) Formulate presentation e) Perform paper prototyping with non-CSE majors f) Install the software tools we plan to use g) Learn the OneBusAway API	1 week	1
3	Zero-Feature Release (due October 28)	a) Add information to our website b) Write separate documentation for users and for developers c) Write the front-end and back-end interface d) Finalize the UI design e) Set up appropriate databases and servers	½ week	1 and 2
4	Beta Release (due November 11)	a) Build the search features b) Implement the forums for bus routes and neighborhoods c) Set up feature to view locations of a certain bus d) Test the major features built e) Revise SRS and SDS f) Update documentation g) Present demo to the class	2½ weeks	1, 2, and 3
5	Feature-Complete Release (due November 18)	a) Implement the remaining major features (setting favorite routes and the help button) b) Build the stretch features as time permits c) Finish testing the app d) Finalize the documentation	1½ weeks	3 and 4
6	Release Candidate (due December 2)	a) Fix all documented bugs and any that may interfere with the user's experience b) Write a code review c) Have three non-CSE majors test and evaluate our app	2 weeks	4 and 5
7	Final Demos	a) Update SRS schedule	1 week	1 - 6

	(due December 9)	b) Write post-mortem document c) Give final demo to the class d) Write individual reflections e) Provide feedback to the instructors		

Team Structure

The project manager of our group is Becky, and she will have the responsibility of making sure we are on schedule. Additionally, she will split the tasks for each milestone among the members of our group. Design decisions will be made through group discussion and a voting process. If a vote results in a tie, we defer to the project manager who will then make the final decision. The front-end team will consist of Becky, Bryce, and Daniel, who will work together on the Android app. David, Daniel, Dylan, and Nick will then make up the back-end team and will develop the API as well as the integration with OneBusAway. Finally, we will have a testing team with Nick, Bryce, and Dylan performing the integration and functional tests. Each member of the group will additionally perform their own unit testing. Our group has been meeting twice a week, from 8:30am to 10:30am on Tuesday and Thursday. We will also schedule meetings in addition to those times when necessary. Lastly, we have been using Discord for text and voice chat when working apart from each other.

What follows is a schedule roughly detailing weekly tasks for each team member.

Week #	Becky	Bryce	Daniel	David	Dylan	Nick	Everyone
4 (ZFR)	Finalize UI, write stubs for basic front end architecture	Develop simple failing Appium test, write stubs for front end	Update website, setup basic Rails architecture	Write stubs for back end, user documentation	Setup test integration with Travis, write stubs for back end	Setup build script, specify FE/BE interface	Developer documentation
5	Implement and test UI stubs, Revise SRS and SDS	Implement and test FE/BE interface stubs (FE), work on system tests	Implement and test FE/BE interface (BE & FE)	Implement and test BE stubs, update user documentation	Implement and test BE stubs, work on system tests	Update build script, implement BE stubs, work on system tests	Begin black box testing
6 (BR)	Finish and test UI, Work on demo	Finish system tests and FE modules	Finalize Rails server code & FE/BE interface	Finish BE, Work on demo	Finish BE, Work on demo	Finish BE, Finish system tests	Finish black box tests

7 (FCR)	Determine viable stretch features	Additional system tests	Finalize documentation	Finalize document ation	Additional system tests	Additional system tests	Feature add-on for relevant modules
8	Work on code review (FE)	User testing	Work on code review (BE/FE)	Work on code review (BE)	User testing	User testing	Bug catching and handling
9 (RC)	Finish code review (FE)	Finish user/syste m tests, review FE unit tests	Finish code review (FE/BE)	Finish code review (FE/BE)	Finish user/syste m tests, review BE tests	Finish user/syste m tests, review system/us er tests	Bug catching and handling
10 (Final)	Update SRS schedule	Work on demo	Work on demo	Post-mort em document	Post-mort em document	Work on demo	Help with post mortem docume nt

Test Plan

Unit testing will be performed on each major class to verify conformity to specification. We will use a combination of black and white box testing. For each module black box tests can be written first by a team member not responsible for its implementation, then white box tests by the implementer. We will use Travis for continuous integration.

For system tests we intend to use stubs and mocks to focus on interactions between the front end (the Android app) and the back end and between the back end and One Bus Away. These tests will be automated with Appium's Java packages. Performance tests will check that our application meets the expectations set in our non-functional requirements (e.g. launch speed and time to display alerts after posting) and possibly include some reasonable stress testing. System tests will primarily be the responsibility of members of the testing subgroup described in the SRS, which includes members working on both the front and back ends.

Usability tests will be performed as the application becomes functional from the perspective of a user. Initially the testers will be only our group members, but as the app becomes more complete we will try to include outside users as well. At first these tests can be done on just our own infrastructure, but at some point we will need to ride buses or wait at stops to test some features. This will potentially be difficult because most of our group members do not have Android phones. Some of our group

members use the bus system daily to commute to and from the university, so once we start usability testing it should be possible for us to do so almost daily.

For all of the basic functionality of our application we expect these tests to be adequate. However it will be almost impossible to test the entire application comprehensively in a real world scenario without a larger user base and well-timed failures of the bus system.

As bugs appear they will be added to Github by the team members who discover them. When they are discovered how to reproduce the bug and how its behavior differs from expectations should be logged. If the member suspected of writing the code causing the bug has difficulty fixing the bug, then they will request that another member look for the bug most likely from the testing subgroup.

Documentation Plan

In order to help users figure out how to use *Where's My Bus?*, we will provide an in-app help section. This will be a FAQ style interface containing detailed instructions on how to utilize the various features of our app. The instructions will include a list of steps which can guide the user through a given use case and additionally may contain images portraying relevant sections of the user interface. The different questions and their accompanying answers can also be grouped together by topic in order to make searching for assistance easier.

Coding Style Guidelines

Ruby: https://github.com/bbatsov/ruby-style-guide
Java: https://google.github.io/styleguide/javaguide.html

There are a few options available to ensure that we follow these style guides. For Ruby there is RuboCop (https://github.com/bbatsov/rubocop). It is a command line program that checks the style of Ruby code based on the guide above. There are also RuboCop plugins available for various text editors and IDEs. Finally, to enforce coding style for Java, we can use the XML code style configuration file provided by Google, in Android Studio. It is likely that these tools do not completely enforce every rule in the above style guides, so additionally, we can review each other's code to make sure it conforms to the relevant coding style guidelines.