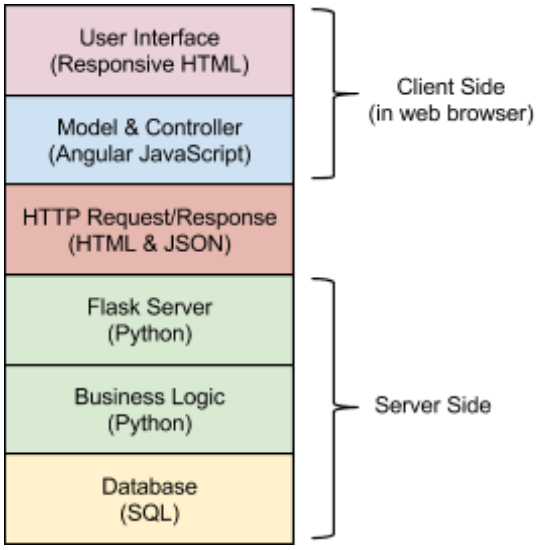
LineUp

System Design Specification

System Architecture

LineUp is a web based application that can be accessed from web browsers on desktops, tablets and phones. Data from user interaction will flow through a series of layers divided into the the client and server sides. Client side code will be written in HTML and JavaScript utilizing the Angular framework. All communication between the client and server will be through HTTP requests and all responses will be in the form of static files (HTML, images, etc.) or JSON data. Server side code will be written in Python using the Flask framework. Lasting data will be stored in an SQL database.



Design Decisions and Alternatives:

1. *Users with accounts vs. temporary users*

We chose to support both users with accounts and temporary users. The alternative was to only allow users with accounts to use the service. This would have required all users to login before being able to join any queues. This decision was made based on input by the original visionaries of the product. Supporting temporary users allows easy user access to join queues quickly. After much debate, we decided that this ease of use outweighs the fact that separate user cases will result in extra use cases, code, complexity, and design.

2. *Storing current queue members in database vs. storing current queue members in memory*

We chose to store current queue members in memory (for now). The alternative was to store current queue members in the database. This would have added much additional database complexity, and additional database writes for every queue modification. For now, since our user base is going to be very small, we can keep queues simply in main memory. This will allow additional speed with less database interaction. We can use software object oriented design rather than having to consider how to put the queues in the database, which will simplify our QueueServer operations and validation checks.

Design Assumptions:

Our design includes a number of explicit assumptions, as well as many assumptions that we are not aware of at this time. As far as explicit assumptions, we assume that users will only feel the need to create one account. There are some additional assumptions being made around account creation. We are currently supporting the use of queues by both anonymous users as well as users with accounts, but at this time we are not making any attempt to preserve data about an anonymous user and later tie that information in to be part of an account that they decide to create.

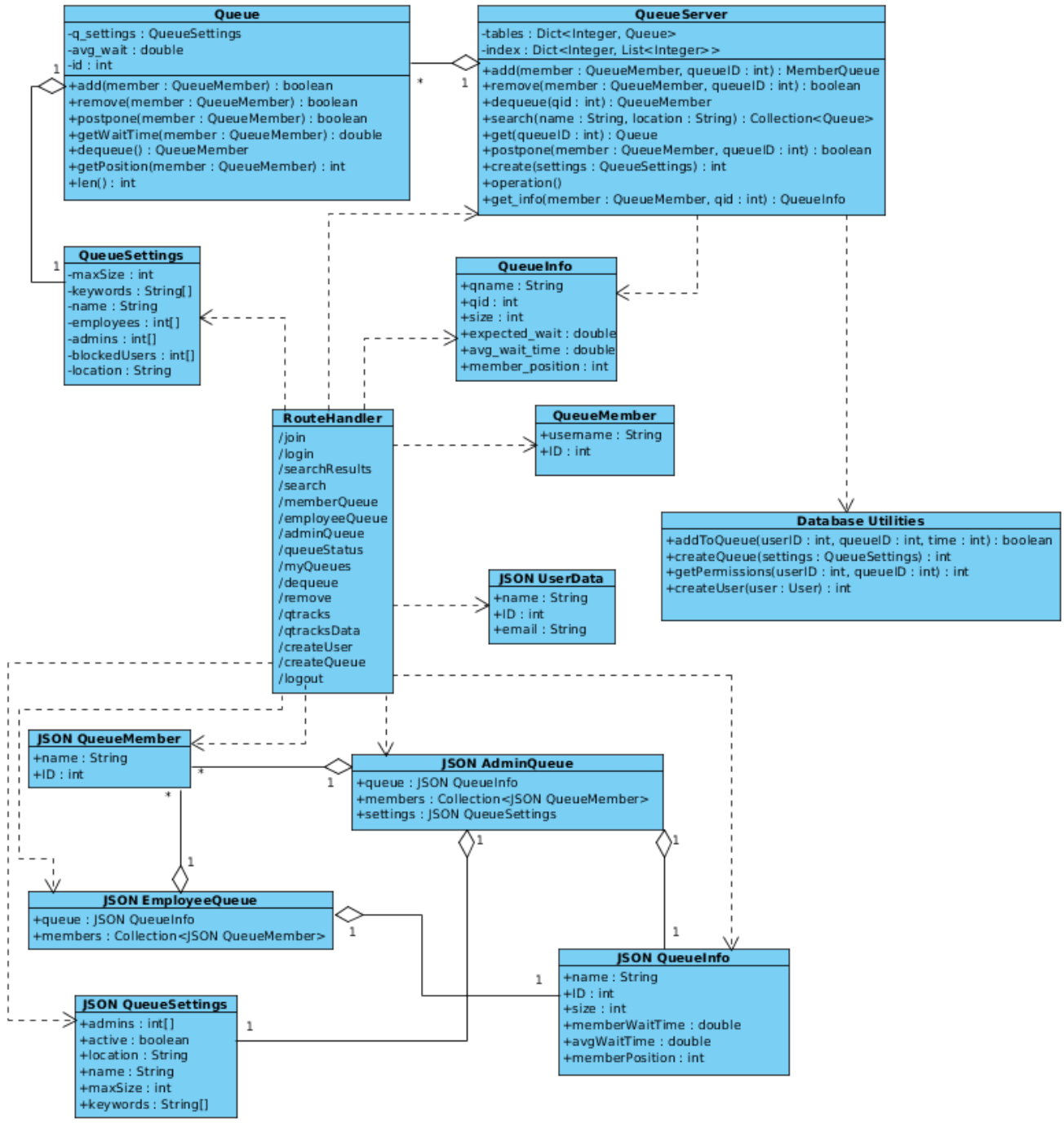
High-Level Data Schema:

LineUp stores multiple data schema in an SQL database to record queue activity and history. Schema includes user accounts, the settings details of a queue, user permission levels associated with specific queues, and historical data for future use in analytic features. We have chosen to store our active queue members in memory. This will be stored in a Dictionary of qid to Queue objects. We will have a reverse index to go from userID to the queues they are in. This will also be stored in memory.This will allow fast access for getting the queues a user is in, and getting a queue by its qid. We chose for our initial release to not support remembering current queue members and ordering in the database because of increased complexity to the database structure and server validation logic. The following is a high level definition for our database schema. The objects mentioned above to store queues in memory are in our object diagrams.

```
Users {
        uid int PRIMARY KEY,
        temporary bool,
        uname varchar(32),
        fname varchar(20),
        lname varchar(20) NULLABLE,
        email varchar(32),
        some function of the password,
}

QSettings {
        qid int PRIMARY KEY,
        name varchar(32),
        max_size int GREATER THAN ZERO,
        keywords varchar(256),
        location varchar(64),
        state bool
}

/* This table stores administrator, employee, and blocked users. */
Permissions {
        id int FOREIGN KEY Users (uid),
        qid int FOREIGN KEY QSettings (qid),
        permissionLevel int
}

/* This stores the history of members of queues. */
QHistory {
        mid int,
        qid int,
        joinTime DateTime,
        exitTime DateTime NULL if still in queue
}
```

Class Diagram:

The representation of many classes on the server side is turned into JavaScript objects when they are passed from the server to the client side. In the following diagram the RouteHandler class provides the translation. (This is implemented by views.py)



Route Interface:

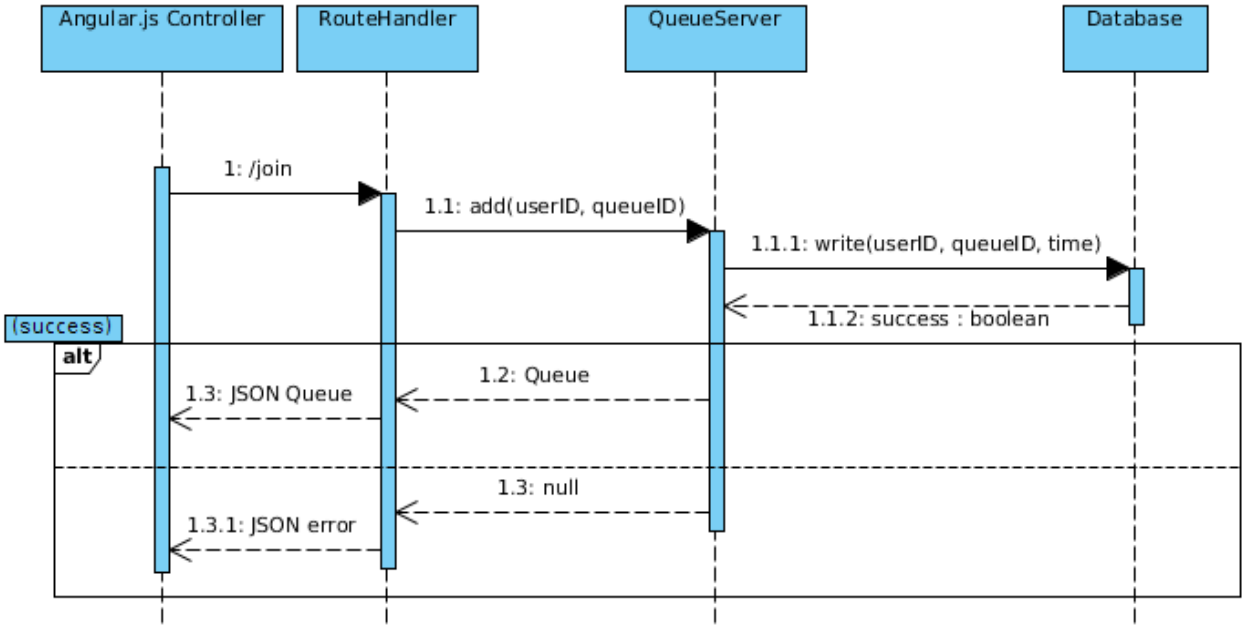
The following URL routes represent the communication interface between the browser in the client side and the server. Requests from the browser are answered with static or JSON responses.

/route	**Method:return**	**Method:return**
/login	GET:Static	POST:JSON UserData
/searchResults	GET:Static	
/search?params	GET:List of JSON Queues	
/memberQueue	GET:Static	
/employeeQueue	GET:Static	
/adminQueue	GET:Static	
/queueStatus?qid	GET:Depending on permissions, one of: JSON AdminQueue, JSON EmployeeQueue, JSON MemberQueue	
/myQueues?uid	GET:JSON Object containing a List of JSON AdminQueue, List of JSON Employee Queue, and List of JSON Queue	
/join?uid,qid,name	GET:JSON Queue	
/dequeue?qid	GET:JSON Queue	
/remove?uid,qid	GET:JSON Queue	
/qtracks	GET:Static (stretch feature)	
/qtracksData	GET:JSON Analaytics (stretch feature)	
/createUser	GET:Static	POST:Static (redirect to login)
/myAccount	GET:JSON UserData	
/createQueue	GET:Static	POST:JSON AdminQueue
/logout	GET:Static (redirect to home)	

Sequence Diagrams:

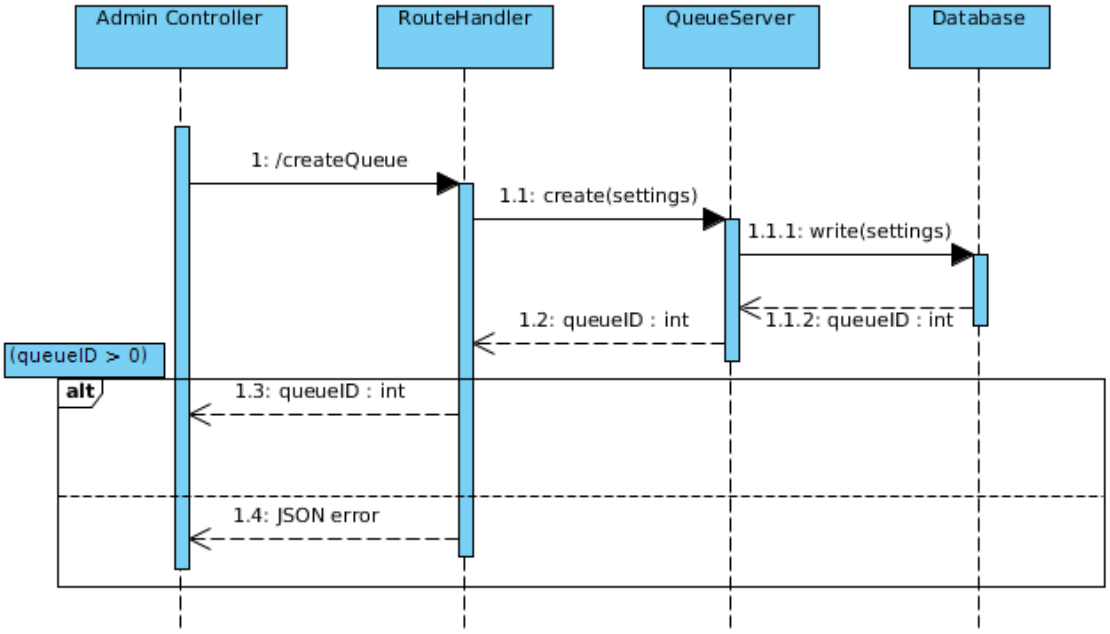
1. Add self to a queue:

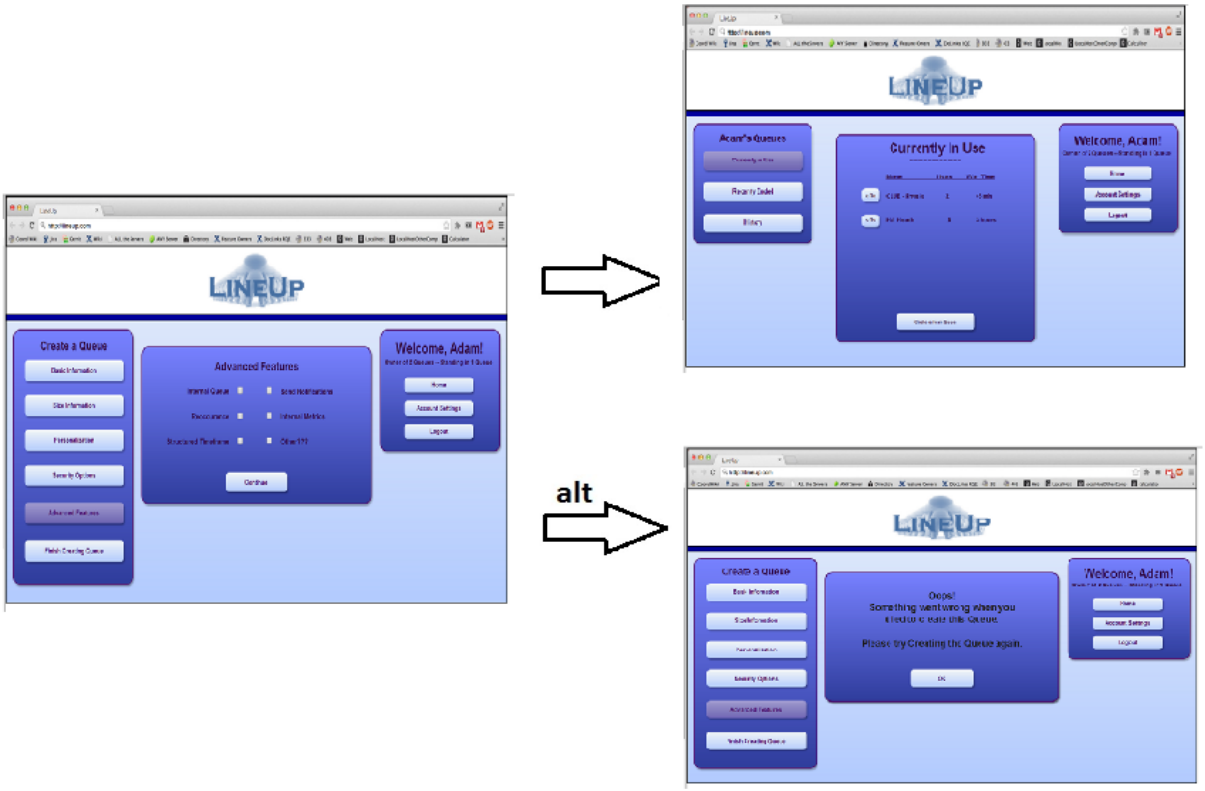
This sequence diagram depicts a user that has already logged in to the service and found a queue that they would like to join. They click the join button to initiate the sequence. The RouteHandler passes the request to the QueueServer, which attempts to write to the database. If this write is successful, the newly updated queue is returned to the RouteHandler, which creates a JSON queue. If the write fails, a JSON error is produced.



2. Admin creates a queue:

This sequence diagram depicts an administrator that is creating a new queue. The admin is already logged in and has entered all of the settings for the queue. They click the add queue button to initiate the sequence. The RouteHandler sends the request to the QueueServer which attempts to add a queue with the specified settings to the database. If the database returns a non-zero queueID (indicating success), the QueueServer adds the queue to its collection and returns the queueID to the controller. If queueID is zero, it returns an error.

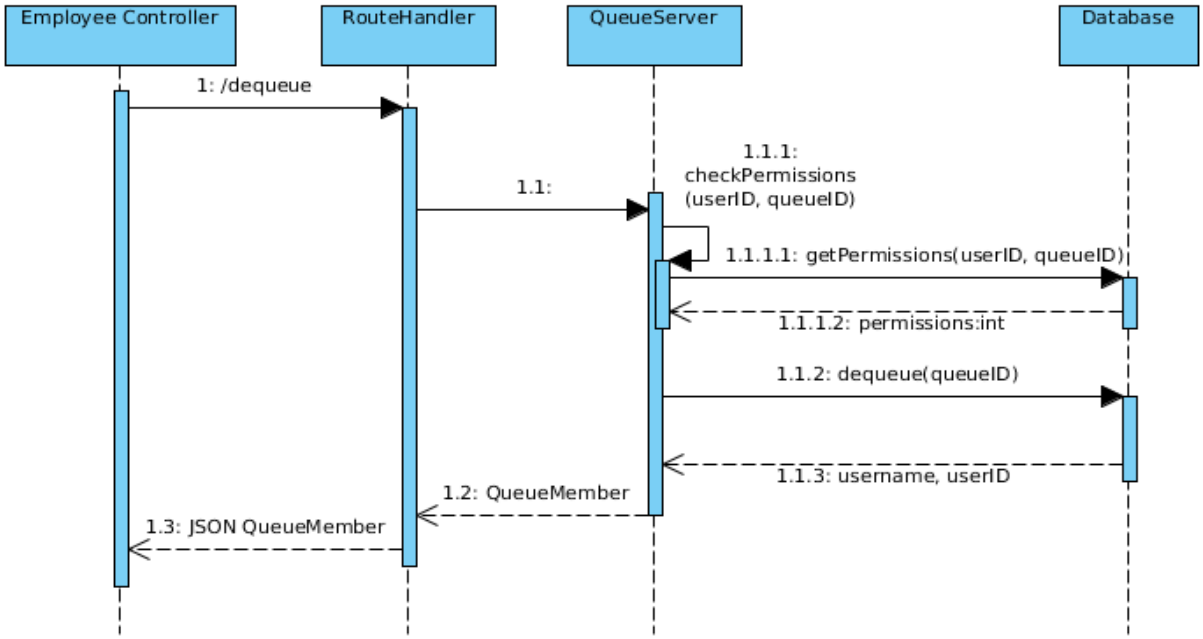


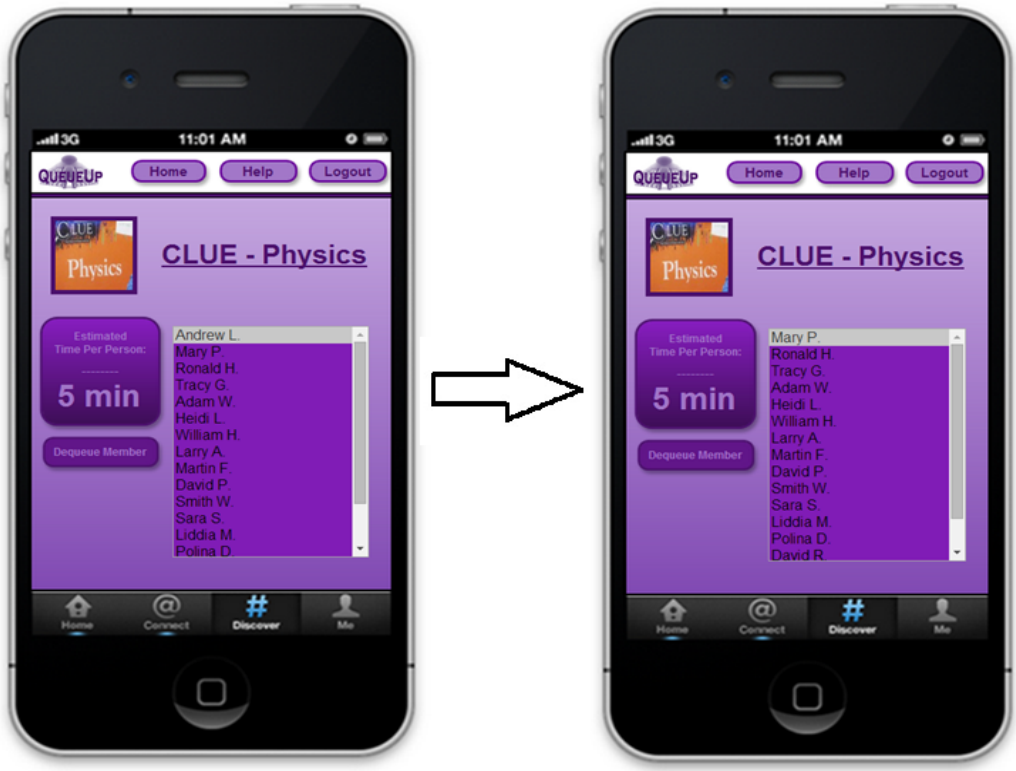


3. Employee Dequeues:

This sequence diagram depicts an employee dequeuing a person from a queue. They are ready to provide services to the customer that is at the top of the queue and the sequence is initiated by them clicking the dequeue button. The RouteHandler sends the queueID to the QueueServer in a request to dequeue the uppermost person. The QueueServer runs a routine to check the permissions which checks the database for the permission level that the current user (the employee) has on the given queue. The QueueServer then queries the database for

the next user and returns the result to the RouteHandler (and also updates its queue).





Process

Risk assessment:

1. *Two Queue Member Types*

Supporting two types of queue members -- those with LineUp accounts and those without -- may prove challenging. We will need different cases for the same requests, depending on if the member has an account or is a temporary user. This has a high likelihood of increasing complexity and causing bugs. During design, it was difficult to reason about how the system would handle a logged in member vs. one that is temporary, however, we did figure out a system for handling it. If problems arise (corrupted queues, corrupted user database), then we will eliminate temporary member features and require all members to have an account. When writing the SRS for this project, we did not realize the complexity that having two user levels would contribute to the design. Now that we have worked hard at sorting out the details of the design, we have a reasonable plan for wrestling those complexities. Nevertheless, we evaluate this to be a medium risk as we will need to continually make sure that all of our code handles both formats of user data. Should the problem become unmanageable during pre-release the impact will be of medium level. We will focus on authenticated users and possibly drop the anonymous member from our build. Should it become troublesome post-release, we will add alerts forcing users to log in.

2. *Choice of Framework*

Choosing to use Python/Flask and Angular is a risk. We have only a recent understanding of these frameworks. There is a medium likelihood that much time will be spent simply learning and understanding the limits of the code. To mitigate this, we are trying to separate the frameworks from the coding. The Web framework and database details will be handled by specific modules in Python, while the queuing logic will be handled by others. As a result, the coding can be done in parallel, and no developer has to learn every part of the framework. This risk has been mitigated somewhat since the SRS, as some team members have already investigated the frameworks and have an understanding of how to implement our project with them, which has allowed us to get better estimates about the impact that this risk presents. We have experimented with a Flask python server prototype with satisfying results. It was easy to setup and understand. A prototype front end is in development using Angular.

3. *Teamwork*

We have 8 people on our team working on various different operating systems. Because of our large group size and intense deadline schedules, this is both a high likelihood and high impact risk. We may face problems with getting everyone set up with the same Python version and environment to do local machine tests. To mitigate, we may have to create a virtual machine image that everyone would install to do development and testing. We have subdivided our team into UI, client-side, and server-side groups, to modularize the work. This risk has changed since the SRS as we now have a better understanding of how our teams will be structured. To see if this risk is materializing, we will use status reports and member evaluations. We will encourage discussing team problems during meetings to solidify our project vision.

4. *Feature Creep*

A novel feature of the app is its wide range of potential users, leading to a risk of how to make it applicable to them without succombing to feature creep. Currently, our risk is low, but without any mitigation, there is a high risk of this occurring. During our meetings, we have already observed that the complexity of the system has grown quite a bit since the initial concept was proposed, leading to different ideas about this risk

since the SRS project. This additional complexity was anticipated and necessary. We may limit our customer range if feature creep occurs. We have stopped talking about new features for the time being and we are defining a concrete schedule which contains a date after which we will not add any more features to the product. To detect whether or not we are keeping this risk level low, we will compare the SRS and SDS documents with the current state of project and ask our clients to test the ease of use of our product weekly. Should the issue rise, however, we will let our PM selectively cut features with client feedback.

5. *Zero-Feature User Accounts*

We would like to include user accounts in our zero-feature release to minimize the amount of backtracking we will need to do to create our alpha and beta releases. No one on the team is sure of how to have secure user accounts, however, making the likelihood of this being an issue high. If creating secure user accounts turns out to be more complex and time consuming than we expect, then we can implement basic accounts without security features during the zero-feature release. Because the zero-release will contain less-sensitive data, the impact of security errors will be low until we get to the later releases. To reduce risk, we have started brainstorming specific security issues, including protecting data when it is stored or transferred, validating requests, auditing data types that are passed to the client side (visible and not visible data), and sanitizing user inputs. We were not fully aware of this risk when we wrote the SRS due to focus on high level details. To detect if this risk materializes, we will try to hack our own server, as well as have validation for every user contact point (receiving data from or sending data to the user).

Project schedule:

Finish Date	Objectives
4/28/14	Getting Started ● Thomas: Presentation preparation + relational diagram for SQL database ● Nora: Web server prototype, static and JSON "Hello World" responses ● Evan W: Web server prototype, static and JSON "Hello World" responses, ensure build can be completed in one step ● Stephen: Angular Multi-Template prototype ● Nick: Angular Multi-Template prototype ● Bryan: Presentation preparation + Angular tutorial + help Thomas with relational diagram for SQL database ● Simone: Complete Bootstrap tutorial ● Evan L: Complete Bootstrap tutorial
5/2/14	Zero Feature Release ● Thomas: Basic Database Utilities (Users, QSettings, Permissions) ● Nora: Route Handler, User Module, test the gitHub bug tracker and make sure it works with our project ● Evan W: QueueServer, Queue, QueueMember. Ensure that deployment is functional; product can be viewed from mobile and desktop platforms. ● Stephen: Build administrator view controllers ● Nick: Build user view controllers ● Bryan: Build employee view controllers ● Simone: HTML Page Layouts/Templates ○ Login, create Admin/Employee/Member Accounts, Create Queue, List of all Available Queues - (In lieu of search results page) ● Evan L: HTML Page Layouts/Templates ○ MemberQueue (for member to see their place in line), EmployeeQueue, AdminQueue (to allow for an admin to dequeue people), QTracks, NewUser, CreateQueue
5/9/14	Beta Commit ● Thomas: Database QueueSettings, handle anonymous users ● Nora: RouteHandler update for anonymous users, user self-removal ● Evan W: QueueSettings, Delete Queue, QueueTypes ● Stephen: Add features to Admin controller for full set of Queue settings ● Nick: Add features to controller for a User removing oneself ● Bryan: UI ● Simone: Polish UI, develop basic testing frameworks for server-side ● Evan L: Polish UI, develop basic testing frameworks for

5/13/14	Beta Test -Thomas: Full implementation of Beta Commit phase component. ● Nora: Full implementation of Beta Commit phase component. ● Evan W: Full implementation of Beta Commit phase component. ● Stephen: Full implementation of Beta Commit phase component. ● Nick: Full implementation of Beta Commit phase component. ● Bryan: Full implementation of Beta Commit phase component. ● Simone: Full implementation of Beta Commit phase component. ● Evan L: Full implementation of Beta Commit phase component.
5/16/14	Beta Release ● Thomas: Code review/shallow testing of Nora code/documentation from Beta Test phase complete. ● Nora: Code review/shallow testing of EvanW code/documentation from Beta Test phase complete. ● Evan W: Code review/shallow testing of Thomas code/documentation from Beta Test complete. ● Stephen: Code review/shallow testing of Nick code/documentation from Beta Test phase complete. ● Nick: Code review/shallow testing of Bryan code/documentation from Beta Test phase complete. ● Bryan: Code review/shallow testing of Stephen code/documentation from Beta Test phase complete. ● Simone: Code review/shallow testing of EvanL code/documentation from Beta Test phase complete. ● Evan L: Code review/shallow testing of Simone code/documentation from Beta Test phase complete.
5/23/14	Feature-Complete Release ● Thomas: QueueHistory, Database QueueAnalytics ● Nora: Ability to change user settings, update RouteHandlers to handle Analytics requests ● Evan W: Postpone, Software QueueAnalytics ● Stephen: Controller for Analytics diplays and User settings ● Nick: Add Queue history and Postpone features to existing controllers ● Bryan: Update SRS, SDS, Architecture Design documents, partial completion of UI test case suite, elimination of major bugs ● Simone: Partial completion of server-side test suite, elimination of major bugs, Software QueueAnalytics ● Evan L: Partial completion of system-wide test suite, elimination of major bugs

5/30/14	Release Candidate ● Thomas: Database bug fixing, user data security. ● Nora: RouteHandlers bug fixes, RouteHandler securiy. ● Evan W: QueueServer bug fixes. ● Stephen: Confirm Javascript works on various devices ● Nick: Confirm Javascript works on various devices ● Bryan: Review all user and developer documentation for publish-readiness, complete UI test case suite. ● Simone: Complete and run server-side test suite. ● Evan L: Complete and run system-wide test suite.
6/4/14	1.0 Release ● Thomas: Verify coverage of Simone's server-side test suite. ● Nora: Brainstorm and execute radical edge case tests. ● Evan W: Find people to do hallway tests. ● Stephen: Verify coverage of Evan L's system-wide test suite. ● Nick: Hallway tests and feedback from random sample of volunteers. ● Bryan: Look over entire project to make sure that deliverable components are presentable. ● Simone: Hallway tests and feedback from random sample of volunteers. ● Evan L: Hallway tests and feedback from random sample of volunteers. ● Everyone at once: Use LineUp at a very fast rate to see how it holds up to high traffic.

Team Structure:

Team Aphrodite is organized according to a semi-hierarchical communion model, with the project manager (Bryan) coordinating communication between non-hierarchical toons. Initially, two toons were formed. Bryan, Evan L, Evan W, and Simone worked on paper prototyping/UI development while Nick, Nora, Stephen, and Thomas began work on the architecture design. Adjusting to the needs of the architecture design assignment led to splitting into three toons. Evan Leon and Simone are working on client-side UI development; Nick, Stephen, Bryan are working on client-side model & controller development; and Thomas, Evan Whitfield, Nora are working on the server-side. Furthermore, each individual team member has an individual specialization as described here:

- Thomas: SQL
- Nora: **Flask** + Python
- Evan W: **Python** + Flask
- Stephen: Angular Javascript
- Nick: Angular Javascript
- Bryan: Project Manager, SQL, Python, Web
- Simone: Web/UI Design & Testing

- Evan L: UI-client-side integration

Milestones in the timeline are elaborated upon here:
4/28/14 - Getting Started: This step will see every team member up to speed with the frameworks we are using. We will also ensure we can build in one step. We have delegated members to present on Monday or Wednesday so that our clients and backers are confident in our ability to deliver.
5/2/14 - Zero-Feature Release: Skeletal implementation of product; version control, bug tracker, deployment functional. We are aiming high for our zero-feature release. We intend to implement support the following use cases on a basic level: Admin creates a Queue, User joins a Queue, Employee Dequeues a person, Create User Accounts, User Login
5/9/14 - Beta-Commit: - 60% check in. This step ensures that progress is being made toward the upcoming beta-release. All components should be implemented at least on a basic level so that it can be completed on the beta-test day and reviewed.
5/13/14 - Beta-Test: 80% check in. All components fully implemented, though not necessarily tested.
5/16/14 - Beta-Release: all major components in place and integrated at basic level. code review and basic testing of existing implementations from the beta-commit is completed.
5/23/14 - Feature-Complete Release - all major functionalities present, user can perform any tasks that are part of requirements; user documentation complete. NO FURTHER FEATURE ADDITION.
5/30/14 - Release Candidate - Bug-free; documents in final form; test coverage verified for completeness.
6/4/14 - 1.0 Release: We will test the system on as large a scale as possible to see if traffic load affects performance and squash any remaining bugs, especially those belonging to edge cases.

Tasks and team members responsible for said tasks are defined in the timeline above.

We are using a listserv for group emailing. We use Google Drive to coordinate unassigned and in-progress assigned documentation. GitHub is our code repository. The GitHub wiki will contain assigned documentation (publish-ready). Team Aphrodite weekly meetings are, at the minimum, Tuesday at 9:30am and Friday at 1:30pm. These meetings are for all team members to discuss their current progress, problems, and upcoming goals. It is expected that team members meet with each other outside of these meetings in smaller groups if necessary to complete their responsibilities. Individual status reports will be posted on the GitHub wiki. Team status reports are the PM's responsibility and will be posted on the GitHub wiki.

Test Plan:

Everyone is free to run their tests as often as they need, but it is required that the code passes all test suites before it is merged into the main repository. All new modules added to the main repository will have a test suite added with them to ensure that there is a place to add tests when needed.

Angular Testing:

There is a variety of testing frameworks available for JavaScript. We have decided to use Jasmine as it is well documented and has received good reviews on many development blogs. It allows for a high degree of

granularity and modularization of the test suites. Additionally, using Jasmine provides the freedom to change development environments as it is IDE independent. Both of the Javascript developers, Nick and Stephen, will work together to identify as many conceivable test cases as possible and create them before developing code for the application. However, Angularjs is a new framework to both developers, so it is acknowledged that many tests will be created throughout development as more becomes known about it.

Server side testing:

Server unit tests will be intended to run whenever the server side developer wants to test a new feature they are working on. These tests should all be automated.
Using python Flask unit testing, make a test for every route and confirm that the data returns is as expected (static page, or JSON data). Testing use cases is very important. We can have unit tests that hit routes in succession in a use case, and confirm the state of the server and the validity of the data returned after the final route. Stress testing will be a part of the unit tests, but could be run separately from the route and use case tests. The unit tests will be run with every daily build. These tests will be written mostly by server side developers, who understand Python. However, the tests should not cross the Route Interface, except to assert that the state of the server is as expected from successive calls through the Route Interface.

System testing:

We will script out different use cases, then require the tester to follow the script exactly and record results. These will not be automated tests. We will plan these use cases with the code in mind, hoping to achieve 95% or better code coverage for the scripts. We should run all system tests two days before every release. Ideally, these would happen much sooner, but time constraints are limiting us. We expect future releases to satisfy all system tests (including past system tests). These tests should build as features are added. The scripts should define a satisfactory final state that the tester will confirm has been reached by the end of the test.
In addition, we will do tests to confirm that the UI works on different devices (mobile, desktop, laptop).

Adequacy of test strategy:

A good test strategy should be easy to run often and include as much automation as possible. This will be the case for the angular and the back-end server testing, but unfortunately, the nature of our system does not allow for the easy creation of automated system and usability tests. Therefore, while those tests will adequately cover the code, they will fall short in terms of ease of use which may discourage running all of them at adequate intervals. Once a week, these scripts will be carried out at one of our meetings.

Bug tracking mechanism and plan of use:

We are using Git version control and Github to store out repository. Github provides an issue tracking tool that we will be using. It is the logical choice as it is sufficiently powerful, is already integrated with our version control system, and allows us to not introduce another, completely separate tool into the development process. The issue tracker provides facility for categorizing bugs, assigning them to developers, and even filtering over those labels. We plan to use all of these features and discuss the state of the issue list at all of bi-weekly team meetings.

Documentation Plan:

Because our product is meant to be used by everyone, of which a vast majority may be non-technical, all user-side documentation should be easily accessible and contain non-technical language. User-side documentation will include an easily-accessible FAQ web page and mouse-over explanatory pop-ups on the actual UI to explain what certain aspects of the product do. Developer documentation will include module specifications and code comments to facilitate black-box testing and debugging, respectively.

Coding Style Guidelines:

Coding style as described by the Google style guides will be observed by developers and will be enforced using validators prior to code reviews. Correctness will be enforced through a single-team-member code review prior to check-in to the repository. Code review tips are located here scientopia.org/blogs/goodmath/2011/07/06/things-everyone-should-do-code-review/.

The style guides and validators are:

- Python with Flask: google-styleguide.googlecode.com/svn/trunk/pyguide.html
    - Prior to code review: run pylint www.pylint.org
- HTML + CSS: google-styleguide.googlecode.com/svn/trunk/htmlcssguide.xml
    - Prior to code review: run through validator validator.w3.org/
- Javascript w/ Angular: google-styleguide.googlecode.com/svn/trunk/javascriptguide.xml
    - Prior to code review: run through validator www.jslint.com/