# Software Requirements Specification

for

# Lyne

Version 1.0.0 approved

Prepared by Joseph Chen, Benson Jin, Di Zhou, Jalen Xu, Daniel Lee, Zhongyuan Tang

CSDS 393 Group 2

February 25th, 2022

## Table of Contents

## Revision History

| Name | Date | Reason For Changes | Version |
|---|---|---|---|
| MVP | 2/25/22 | Minimum viable SRS | 1.0.0 |
| | | | |

# 1. Introduction

## 1.1 Purpose

Lyne Version 1.0 is a full stack web application to make lining up for restaurants, clinics, and other businesses easy and safe through a simple scan of a QR code and to make managing those users easy for those business owners. This software requirements specification will cover both the frontend and backend portions of the application, but will not go into detail for the website deployment process. The continuous integration and continuous deployment pipelines will also not be covered.

## 1.2 Document Conventions

Requirement priorities will range from: Low, Medium, High, Critical. Critical means that the application cannot be shipped if the requirement is not met. Requirements should follow the REQ-NUMBER format and sub-requirements should inherit the encapsulating requirement (usually a user story).

## 1.3 Intended Audience and Reading Suggestions

Developers, project managers, testers, and documentation writers should read this SRS. Developers should be familiar with the functional requirements because they contain plans for component implementations. Product managers and testers should pay attention to section 2 and the user stories in Section 3/4. They contain critical context for properly building and integration-testing the application.

## 1.4 Project Scope

Many businesses have limited the occupancy in their in-person locations due to social distancing mandates and state regulations. As such, high traffic locations, such as COVID clinics, restaurants, and hospitals, often have their customers wait outside in lines that are not properly socially distanced. This leads to an increase in the spread of contagious diseases, such as COVID-19, and increases the satisfaction of customers who need to wait outside.

Lyne is a web application that should make it easy for business administrators to manage the users in their queue and make it easy for users to join queues for a business. Businesses can provide a better user experience for their customers, may experience less complaints, and provide a safer working environment.

## 1.5 References

The code and relevant documentation will be open-sourced at https://github.com/jchen42703/waiting-line-app.

# 2. Overall Description

## 2.1 Product Perspective

Lyne is a standalone full stack web application being built "from scratch." Lyne is currently in development, and many requirements have already been prototyped at the open source link in **1.5.** The SRS will cover the code architecture, but will not cover DevOps operations (continuous integration/continuous delivery) and infrastructure.

## 2.2 Product Features

The major features that the application will have are:

1. Logged-in administrators can manage queues.
    a. "Manage" means that the administrator (a business owner or clinic owner) can create queues, control when queues start, delete queues, and access queue information from a protected dashboard unique to their account.
2. Logged-in administrators should see a specialized dashboard for each queue.
    a. The administrator should be able to navigate to a special page for each queue that he/she created and see relevant metadata and analytics.
3. Logged-in administrators can ban users from joining a queue.
    a. Administrators can ban certain emails and phone numbers from being able to join their queue.
4. Logged-in administrators can manage the users in a queue and see relevant information.
    a. The administrator should be able to see a list of users that are currently and have been in the queue in the past. The admin should be able to ban, delete, or "pop" users from the dashboard. "Popping" the user notifies the user that it's their turn and moves the queue forward.
5. Administrators can sign in with only their Google/Facebook accounts.
6. Users can scan a QR code to join the queue.
    a. Scanning the QR code redirects them to a form on our website to submit their email and phone number to join the queue. We may add bot protection, such as ReCaptchav3 to prevent abuse.
7. Users see their queue status after joining a queue.
    a. Once they have successfully joined the queue, users should be able to see their place in line, business/clinic information, estimated waiting time, and the information they registered with.
    b. Once the admin "pops" the aforementioned user, the user will be notified and they should be able to then go to the business location and enter safely!

The details for the above features are described in Section 3.

## 2.3 User Classes and Characteristics

### *2.3.1 Administrators*

Administrators are the business owners, and clinic clerks/managers that will manage queues and how customers will enter their property premises. Features 1-5 in **Section 2.2** describe how the administrators will use our product. Unlike users, administrators will need to be authenticated.

We expect that administrators will not be very tech savvy. Hence, we chose to use OAuth 2.0 for authorization to be secure and make the initial learning curve easier for users. Additionally, we want to make the user experience on the dashboard a critical priority because all of our revenue will be generated from administrators being willing to pay for our services. A key part to a good user experience is simplicity. We don't need to overload the dashboard with too many features because that could make using the application too complicated and would deter new administrators from signing up for Lyne.

#### *2.3.2 Users*

Users will be the customers, patients, etc. that are joining the queues to be able to access the administrator's services. Users will be a general audience depending on the business. For example, restaurants, such as bubble tea shops, will have a younger audience that are familiar with technology and how to use QR codes. On the other hand, clinics may experience an older "user" population that are not as tech-savvy. They may have trouble signing up for queues using QR codes, so giving the administrators the ability to manually add users to the queues is critical to making the application accessible to a wide audience.

## 2.4 Operating Environment

Lyne will be a web application designed to run on modern browsers. However, since we are building the application in Typescript, we can also provide support for older browsers (i.e. Internet Explorer) by compiling to CommonJS instead of ES6.

## 2.5 Design and Implementation Constraints

We should avoid using too many external dependencies to build Lyne to avoid long build times and a laggy application. We expect the dashboard to be performant on all devices, so it should not require GPU or too much CPU. **React Router v6** will be used to render Lyne as a single page application, which will make the app run smoothly.

Security is a top priority because administrators may not be tech-savvy and follow security best practices. Hence, OAuth (Sign in with Google/Facebook) makes the administrator registration process easy and secure.

Additionally, all of the code should be written in Typescript (v4.5+) to allow for consistency across the frontend and backend. All shared data transfer objects should be located in a shared repository as either types or interfaces: https://github.com/jchen42703/waiting-line-app/tree/main/packages/shared-dto. Likewise, all unit tests should be done with Jest (v27.4.7+) and TS-Jest (if needed).

## 2.6 User Documentation

Developer documentation should be done in-code, following the JSDoc format. Related design documentation should be located at https://github.com/jchen42703/waiting-line-app/tree/main/docs.

Documentation for app usage for administrators should be located on the website home page. There should be a separate **/help** route/views and related subroutes/views for the tutorials on how to do certain actions, such as managing queues, creating QR codes, etc. All documentation should be done in the frontend. If we scale and add more complex features, adding an FAQ section or a forum could be helpful.

## 2.7 Assumptions and Dependencies

We assume that the frameworks (React, Express, etc.) are working properly. We also assume that Google and Facebook are not going to deprecate the usage of their OAuth 2.0 APIs. All dependencies should be fixed according to the versions located in the package.jsons:

- https://github.com/jchen42703/waiting-line-app/blob/main/package.json
- https://github.com/jchen42703/waiting-line-app/blob/main/packages/backend/package.json
- https://github.com/jchen42703/waiting-line-app/blob/main/packages/frontend/package.json

Each person should only be using Node v16 LTS to keep the package-lock.json files consistent as well.

We also assume that the administrators have Internet access and a Google/Facebook account. Users that don't have a phone or device that can scan a QR code will also face troubles receiving queue alerts.

# 3. System Features

## 3.1 Registered admins can manage user queues.

### 3.1.1 Description and Priority

**Priority: Critical**
Registered administrators should be able to manage queues that users can join. The "manage" operations are **creating queues, deleting queues, and updating queue information.** The queue information should be visible and easily accessible on a dashboard.

### 3.1.2 Stimulus/Response Sequences

- Administrator signs in to their account and should be able to create a queue from the navigation sidebar by clicking on a + button. This should create a new queue button that lets the user navigate to a personalized queue page.

- On the same dashboard, the administrator should be able to delete multiple queues by clicking on a garbage can button. This will toggle a X mark on all queue button elements present and users can click on multiple X buttons for each queue to delete the queue.

- When an administrator clicks on a queue button, they will be navigated to the queue's dashboard. The administrator should be able to update queue information (queue name, icon, description) by clicking on a pencil button on that page. The click will pop up a modal that contains a form with information that the administrator can edit.

### 3.1.3 Functional Requirements

*REQ-1: Dashboard /dashboard; Landing Page*
There should be a base **Dashboard** React TS component when the administrator signs in successfully. The component should display a welcome message and **QueueList component,** which renders a list of queues **only they created.** If they haven't created any queues, display a **NoQueue** component that displays "No queues created." **QueueList** should call a GET request to **/api/admins/queues** which will return a list of **Queues** the admin has created. Each **Queue** should be a JS object with the **queueName, description, timeCreated (UNIX timestamp in ms), liveDate + time (UNIX timestamp in ms), closeDate + time (UNIX timestamp in ms), repeatCycle (one of "daily", "weekly", "monthly", or null)** attributes.

The popup form modal should be a separate component: **CreateQueueModal.** It should do proper validation (checking types, malicious strings, etc) before sending the POST request on submit. If the user inputs improper data, then the form should not submit any requests and should inform the user of any fields that have mistakes.

*REQ-2: Dashboard /dashboard; Create Queue*
There should also be a **Button** that toggles an **onClick** callback to popup a ChakraUI modal to create a queue. The modal should contain a form and a SUBMIT button. The onSubmit callback should POST to **/api/queue/create** with the admin's **adminId** and associated metadata. Currently, the metadata should be: **queueName, description, timeCreated (UNIX timestamp in ms), liveDate + time (UNIX timestamp in ms), closeDate + time (UNIX timestamp in ms), repeatCycle (one of "daily", "weekly", "monthly", or null).** Other properties can be added down the line as needed. The form should have input validation to check that inputs are correctly formatted and are of the right types. If they are not, the form should show warnings and not call the POST request.

Only when the server returns 200 and a **queueId** can a **QueueButton** be created in the **QueueList** component. These buttons should route to **/dashboard/:queueId** when clicked. If the server returns any other status code, raise a **toast** that shows a message (i.e. "Something went wrong!") and to show the associated server error message.

*REQ-3: Dashboard /dashboard; Delete Queue*
There should also be a garbage can **Button** that toggles the **canDelete** state in **QueueList**, which toggles the **canDelete** states in each **QueueButton.** Each **QueueButton** will then display an X button. Once a user clicks on this X button, this will toggle callback to remove the deleted queue's **queueId** from a managed list of **queueIds** in **QueueList**. Once the queueId is removed, this should trigger a rerender and the associated queue button should be removed. The aforementioned callback should also send a POST request to **/api/queue/delete** with the **queueId** attribute in the body and the authentication cookies. The server will then delete the associated queue from the database.

*REQ-4: Queue Dashboard /dashboard/:queueId*
**/dashboard/:queueId** should be a dynamic route based on the **queueId**. It should then render a **QueueDashboard** component based on the **queueId** and the logged-in administrator.

There should be a **pen** button that toggles the metadata elements displayed on the page. Users should be able to type and edit those components. When the user presses ENTER, the components should send an UPDATE request to **/api/queue/update** with the **queueId** and **adminId.**

## 3.2 Logged-in administrators should see a specialized dashboard for each queue.

### 3.2.1 Description and Priority

**Priority: Critical**

Administrators should have a unique identifier for their account. All dashboards should utilize this identifier when the administrator is logged-in to fetch that specific administrator's data.

### 3.2.2 Stimulus/Response Sequences

- Administrator logs into their account.

- They should see queues only they created.

*REQ-1: Administrators must have an unique identifier.*

When an administrator registers, they must be assigned an **adminId.** This should be generated using Javascript's native **crypto.randomUUID** function because it is cryptographically safe and fast.

*REQ-2: **adminId** should be parsed from authentication cookies to get the admin data.*

Whenever a logged-in administrator interacts with a component that sends an HTTP request, it should send over an encrypted authentication cookie that contains an "**adminId: string"** key value pair. The backend server should then parse the **adminId** from the cookie and use that to query the database and get personalized administrator data.

## 3.3 Logged-in administrators can ban users from joining a queue.

### 3.2.1 Description and Priority

**Priority: Middle**

Administrators can ban users from joining a queue. When they ban a user from a queue, that administrator has access to a list of blacklisted emails and numbers. When a banned user tries to join a queue it will notify them that they are banned.

### 3.2.2 Stimulus/Response Sequences

- Administrator bans a user from a queue

- User should not be able to join that queue anymore

*REQ-1: Queue Dashboard /dashboard/:queueId/ban; Ban user*

An administrator must have an encrypted authentication cookie, "**adminId: string"** key value pair, in order to ban a user. When an administrator is in a dashboard they can ban a user from a queue by providing a phone number and email.

*REQ-2: Queue Dashboard /dashboard/:queueId/bannedUsers; Administrators must have a list of banned emails and phone numbers .*

When an administrator logins in they should be able to see the emails and phone numbers of banned users. When administrators ban a user from a queue they are added to this list.

## 3.4 Administrators can manage the users in a queue.

### 3.4.1 Description and Priority

**Priority: Critical**

Administrators should be able to "call" the next user in the queue by clicking a **Button**. This should pop the user data from the queue and transfer it to a separate list **PreviousUsers** that tracks previous users. The user that was called should receive a notification that they have been called. In addition, there should be a toggleable **Button** that allows the administrator to remove users from the queue, no matter the position, without calling them.

### 3.4.2 Stimulus/Response Sequences

*Next User:*

- On the second admin dashboard page, the administrator can click the Next User button.
- Once the Next User button is clicked, the user at the very front (i.e. first place) in the queue is popped.
- The user that was popped receives a notification that they have been popped (e.g. "[The administrator] is ready for you!").

*Delete User:*

- On the second admin dashboard page, the administrator can click the Delete User button.
- Once the Delete User button is clicked, additional buttons are displayed next to each user in the queue. Clicking the Delete User button again will return the dashboard to its original state.
- When the administrator clicks on these new additional buttons, a modal is displayed that confirms whether or not the administrator wants to delete the user corresponding to the button.
- If canceled, the modal disappears and the administrator goes back to the dashboard with the delete buttons toggled on.
- If confirmed, the user is removed from the queue and notified with a message (e.g. "You have been removed from the queue."), and the administrator goes back to the dashboard with the delete buttons toggled off.

*REQ-1: Queue Dashboard /dashboard/:queueId; Next user*

There should be a **Button** that pops the user at the front of the queue (i.e. the user that is first in the queue). When the button is clicked, it should send a request to the server to add the information of the user at the front of the queue to the **PreviousUsers** list, then to remove the user from the queue. It should also trigger a re-render of the **UserTable** to represent the new list of users.

*Queue Dashboard /dashboard/:queueId; Delete user*

There should be a **Button** that toggles the **canDeleteUser** state in the QueueTable component (both on and off). This rerenders the **UserTable** so that buttons are visible to the right of each user row in the table. Clicking on one of these buttons should display a **ConfirmUserDeletion** modal to confirm or cancel the removal of the corresponding user from the queue. Confirming the deletion should hide the **ConfirmUserDeletion** modal and send a request to the server to remove the user from the queue and trigger a re-render of the **UserTable** to represent the new list of users. This should also notify the removed user that they have been removed from the queue and toggle the **canDeleteUser** state off. Canceling the deletion should hide the **ConfirmUserDeletion** and keep the **canDeleteUser** state.

## 3.5 Administrators can sign in with Google/Facebook.

### 3.1.1 Description and Priority

**Priority: Critical**

Existing and new administrators can create/login via Google or Facebook. The information received from Google or Facebook is used to add the administrator to the database and authenticate them and prevent unauthorized access.

### 3.1.2 Stimulus/Response Sequences

- Administrators sign in to their accounts through either Google or Facebook. If it is an existing administrator then it is checked against the database and authenticated. Otherwise, new administrators are added to the database and authenticated.
- Once authenticated and logged in, the administrator session is stored in the database and the server stores the session id in a cookie which is used to authenticate requests.
- When an administrator logs out they no longer have access to the dashboard and must be re-authenticated by logging through Facebook or Google.

### 3.1.3 Functional Requirements

*REQ-1: Login /login; Login Page*

There should be a base **Login** React TS component when the administrator goes to login or signup that takes them to a dashboard for managing their queues.The component should display a sign-in with Google and sign-in with a Facebook button. With the use of the Passport library, after clicking on either the sign-in button for Google or Facebook administrators will be brought to a page where they can choose the respective Google or Facebook account that they want to login into the dashboard with.

*REQ-2: OAuth 2.0 /auth; Authentication with Google or Facebook*

If the administrator is new then they are added to the database and then authenticated. Otherwise, we check against the database and authenticate the administrator. When the administrator logs in and grants permission, the Google Authorization Server sends the backend server an access token along with a list of scopes of access granted by the token. If the administrator does not grant permission then the server returns an error. Then the scopes in the access token response are checked to make sure that the email and name are included to allow for dashboard functionality. After our backend obtains the access token, it is sent to the Google API in an HTTP Authorization request header. The access tokens have limited lifetimes so we will need to obtain a refresh token in order to obtain new access tokens. For both cases, we finally store the administrator state as a session in our database and store the session id in the cookie's browser. This makes sure that the administrator is logged in even if switching to a different route. All of this is handled through the Passport library.

*REQ-2: Logout /logout; Logout of administrator account.*

In the dashboard page, there will be a logout button that allows administrators to logout. Upon clicking the logout button, passport terminates the login session, removing the req.user property and clearing the login session (if any). Now the administrator must login again and re-authenticate in order to access the dashboard.

## 3.6 Users can scan a QR code to join the queue.

### 3.6.1 Description and Priority

**Priority: Critical**

Users should be able to sign up to join a queue by scanning a QR code. The sign-up form includes the user's **name**, **email**, and **phone number**.

### 3.6.2 Stimulus/Response Sequences

- A user scans the QR code of a queue and is redirected to the user signup page of that queue.
- The user fills out the user signup form by entering their name, email address, and phone number into the input boxes.
- When the user clicks on the join button under the user signup form, they will join the queue and be redirected to their waiting page. If the user enters invalid input, they will see a warning on the page and will not be able to join the queue.

### 3.6.3 Functional Requirements

*REQ-1: User Signup Page /users/:queueId; User Signup Form*

There should be a **UserSignupPage** React TS component when the user is redirected by scanning a queue's QR code. The component should display a welcome message and a **UserSignupForm** component. The **UserSignupForm** contains three input fields (**name**, **email**, and **phone number**) and a **Join** button. Users should be able to enter their information into the fields. Only when all the fields are filled with valid input can the user join the queue by clicking the **Join** button. When an input is empty or is invalid, there should be a corresponding reminder under the input box (i.e. "Please fill out this field" or "Please enter a valid email address" or "Please enter a valid phone number").

*REQ-2: User Signup Page /users/:queueId; Join Queue*

The **Join** button should toggle an **onSubmit** callback to POST to **/api/queue/join** with the queue's **queueId** and associated metadata. The metadata should include the user's **name**, **email**, and **phone number**. The **Join** button toggles the callback only when all fields in the **UserSignupForm** are filled out with valid values. When the server returns a **userId**, the users should be redirected to **/users/:queueId/:userId**, which is the **User Waiting Page.** If the server returns an error, a **toast** will appear at the top of the page, showing the error message (i.e. "Woops! Looks like something went wrong with our servers. Please try again. ").

## 3.7 Users can see their queue status after joining a queue.

### 3.7.1 Description and Priority

**Priority: Critical**

Users who joined the queue should be able to see their queue status and sign-up information,. They should also be able to exit the queue. The queue status includes **place in the queue**, **total number of people in the queue**, **estimated waiting time**, **time joined**, and **time elapsed**. The sign-up information includes the user's **name**, **email**, and **phone number**.

### 3.7.2 Stimulus/Response Sequences

- Users join a queue and should be able to see their queue status and sign-up information. The users should be able to get a new status every minute.
- Users should be able to exit the queue by clicking on an **Exit** button. The click should pop up a modal which contains a confirmation message, an X button on the top and two buttons on the bottom: **Yes, exit the queue** and **Stay**.
- If the user clicks the **Yes** button, they will exit the queue. If the user clicks the **Stay** button or the X button, they will stay in the queue and the modal will be closed.

#### 3.6.3 Functional Requirements

*REQ-1: User Waiting Page /users/:queueId/:userId; User Waiting Page*
There should be a base **UserWaitingPage** React TS component when the user joins the queue successfully. The component should display a **QueueStatus** component and a **UserProfile** component. The **UserProfile** component should be a modal at the bottom of the **UserWaitingPage.** It should display a title (i.e. "User Profile"), a **UserProfileBox** that contains the user's information, and an **Exit button**.

REQ-2: *User Waiting Page /users/:queueId/:userId; Queue Status*
The **QueueStatus** component should display a progress bar, **user's place in the queue**, **total number of people in the queue**, **estimated wait time**, **time joined**, and **time elapsed**. The component calls a GET request every minute to **/api/queue/progress**[ZD1] with the queue's **queueId** and the user's **userId**. The GET request should return the queue status, including the user's **place in the queue**, **joined time**, the queue's **total number of people** and **estimated wait time per user**. If the user is the first person in the queue, the **QueueStatus** should display a message (i.e. "It's your turn!") under the progress bar.

Only when the GET request does not return an error can the **QueueStatus** display the user's queue status. Otherwise, it should display an error message.

*REQ-3: User Waiting Page /users/:queueId/:userId; User Profile*
The **UserProfileBox** component should display the user's **name**, **email**, and **phone number**. The component calls a GET request with the **userId** to get the registration information.

*REQ-4: User Signup Page /users/:queueId; Exit Queue*
The **Exit** button should toggle an **onClick** callback to pop up a confirm dialog. The confirm dialog should display a confirm message (i.e. "Are you sure you want to exit?"), a **Yes, exit the queue** button, and a **Stay** button. The **Yes** button should toggle an **onClick** callback to remove the user from this queue, and the **Stay** button should toggle an **onClick** callback to close the confirm dialog.

# 4. External Interface Requirements

## 4.1 User Interfaces

UI-1: The system will provide the administrator with a means of creating, deleting, or editing a queue.

UI-2: The system will provide the administrator with information on each queue: the queue name, number of users in the queue, the users in the queue, estimated wait time, and the previous users from the queue.

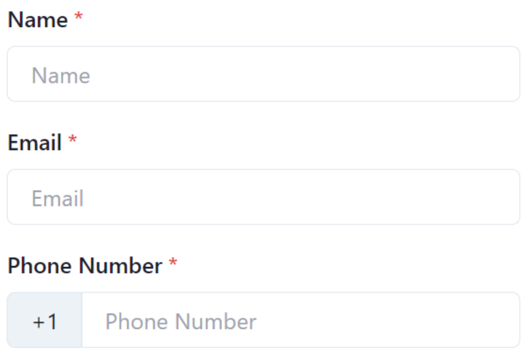
UI-3: The system will provide the administrator with a means to pop the next user in the queue or to delete a user from the queue.

UI-4: The UI will scale appropriately with different window sizes and screen sizes.

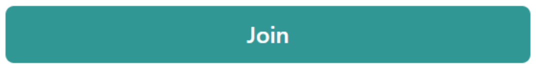
UI-5: The UI will provide a way of easily differentiating both queues and users from one another.

UI-6: The UI will have a step to confirm any deletion.

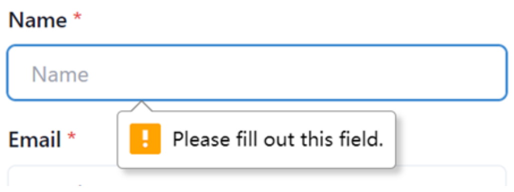
UI-7: The system will provide the user with a registration form with three input fields: name, email, and phone number.



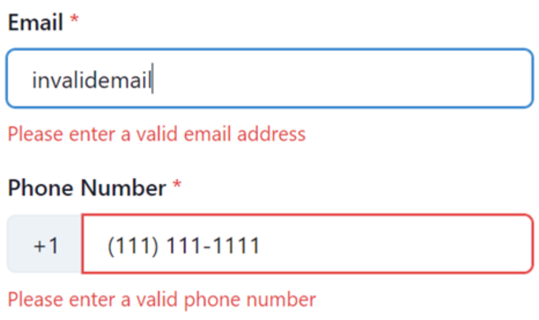
UI-8: The system will provide the user with a join button to enter the queue.



UI-9: The user will see a warning if they click on the join button before filling out all the fields.



UI-10: The user will see a warning if they entered invalid input and click on the join button.


Email *

invalidemail

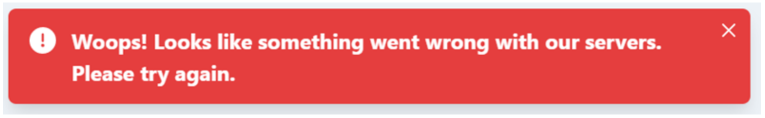
Please enter a valid email address

Phone Number *

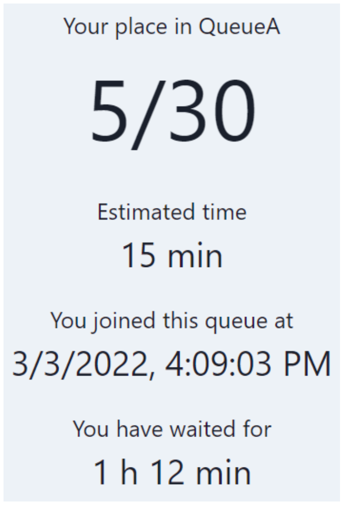
+1 (111) 111-1111

Please enter a valid phone number

UI-11: The user will see an error message if they cannot join the queue due to a server problem.


Woops! Looks like something went wrong with our servers. Please try again.

UI-12: The system will provide the user with their place in the queue, the total number of people in the queue, the estimated waiting time, their joined time, and the elapsed time.


Your place in QueueA

5/30

Estimated time

15 min

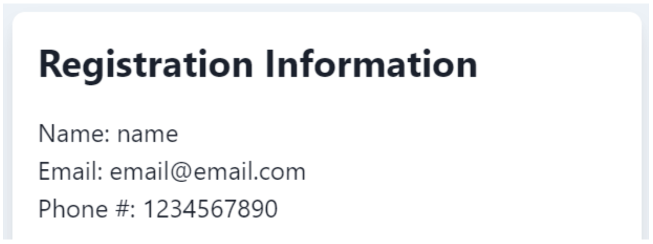
You joined this queue at

3/3/2022, 4:09:03 PM

You have waited for

1 h 12 min

UI-13: The system will provide the user with their registration information: name, email, phone number.


Registration Information

Name: name

Email: email@email.com

Phone #: 1234567890

UI-14: The system will provide the user with an exit button.

UI-15: The user will see a confirmation dialog if they click on the exit button.

UI-16: The system will provide the user with a yes button and a stay button in the confirmation dialog.

## 4.2 Hardware Interfaces

The web app will run on both mobile and desktop browsers. Since all of the Typescript will be compiled to `ES5/commonjs`, the web app will work on almost all browsers. Only the QR code functionality will require user access to a camera and a QR code scanner.

The web app will be hosted on AWS EC2 and deployed with `nginx` to act as a reverse proxy and load balancer.

## 4.3 Software Interfaces

All of the code is deployed on Github and open sourced at https://github.com/jchen42703/waiting-line-app. As such, **git 2.25.1** should be used for version control. Additionally, the code is deployed as a Javascript/Typescript (^4.5.5) monorepo using **lerna 4.0.0** and **yarn 1.22.x** workspaces. This lets the frontend and backend share the same interfaces (**@waiting-line-app/shared-dto**) for request/response payloads without having to compile any Javascript from Typescript. The main data interfaces are located here. For each subpackage, Prettier and ESLint are employed to maintain code quality. Tests are deployed using **jest.** The backend uses **ts-jest** as the preprocessor to conveniently run tests.

The client is built using Facebook's **React (^17.0.2)** framework and Typescript. **ChakraUI (^1.8.3)** is the chosen UI components library, which will speed up development. **react-router-dom (^6.2.1)** is used for setting up routes and allows us to render the dashboards as a single page application (SPA). Also, for convenience, the React repository is configured to use both **scss (sass ^1.48.0)** and **tailwind (tailwindcss ^3.0.21)** to make styling quicker and easier.

The backend server is built with **Express 4.** The MongoDB ORM is **mongoose 6**, which lets us easily create schemas and query our MongoDB database (currently hosted on MongoDB Atlas). **pino 7** is used for logging because it is much faster than its competitors (i.e. **winston**). Various **passport** modules are used in conjunction to implement OAuth 2.0. We do not plan on supporting traditional username/password authentication, and instead are relying on OAuth 2.0 authorization to give administrators access to their dashboards. The specific package versions are located here.

### 4.3.1 HTTP Routes

*REQ-1: POST /api/queue/create*

This endpoint is intended to create a queue for a logged-in administrator. The endpoint takes in an authentication cookie and the server parses the **adminId** from the cookie. The **adminId** is checked whether it exists by examining if it's a string or if the request contains an **adminId**. After checking if the **adminId** exists, then the endpoint checks if it is a valid **adminId**. If none of these pass, then the server returns an error status code 400 and an error message. The endpoint creates a new **Queue** model that contains a **queueId**, **adminId, canJoin** boolean, and a **queue** (using an array data structure)**.** It returns a **queueId** for that **adminId** that was created for the **Queue** model. This endpoint essentially creates a queue for the admin.

*REQ-2: POST /api/queue/join*

This endpoint takes in a request body containing a **queueId** in a JSON format. The **queueId** is checked whether it exists by examining if it's a string or if the request body is undefined. If it doesn't pass, then the server returns an error status code 400 and an error message. The endpoint creates a new **user** model that contains a **userId** and **initQTime** (initial queue time). This is then added to the **queue** according to the given **queueId** in the request body. This is done by searching for the **queueId** in the database and pushing the **user** into the **queue**. If it's successful, then the server returns a response body {userId: userId}. If it isn't successful, then the server returns an error status code 500 with a message "join failed." If the **queueId** is invalid, then the server returns an error status code 400 and an error message "bad queueId." This endpoint essentially adds the user into the queue.

*REQ-3: POST api/queue/pop*

This endpoint takes in a request body containing a **queueId** and an authentication cookie. The server parses the **adminId** from the cookie. The **queueId** is checked whether it exists by examining if it's a string or if the request body is undefined. If it doesn't pass, then the server returns an error status code 400 and an error message "queueId must be a string." The endpoint searches for the given **queueId** in the database. Once found, the server must then check that the **adminId** associated with the queue matches the **adminId** from the authentication cookie. If correct, the endpoint pops the first user in that specific queue. If successful, the server returns a response body that contains the popped **userId** in the queue. If it's not successful, then the server returns an status error code and an error message. This endpoint essentially pops the first user in the queue.

*REQ-4: UPDATE /api/queue/update*

The endpoint should be used to update queue metadata (i.e. **queueName, description, timeCreated (UNIX timestamp in ms), liveDate + time (UNIX timestamp in ms), closeDate + time (UNIX timestamp in ms), repeatCycle (one of "daily", "weekly", "monthly", or null)).** The request body should have **queueId** and the aforementioned queue attributes. The request should have a properly authenticated cookie that is sent along with it. The route in Express should parse that request body and authentication cookie. Then, it should update the document in the **queues** collection in the MongoDB database that corresponds to the **queueId.** If a document is not found with the **queueId** and the parsed **adminId** from the authentication cookie, then the server should return a 400 with a response body of { message: "invalid queueId" }. If the database query fails, then the server should return with a 500.

*REQ-5: GET /api/queues/all*

Gets a list of all users for a specified **queueId.** In other words, the request query should contain the **queueId** attribute and the request should contain an authentication cookie. Once the request passes through the authentication middleware to validate the authentication cookie, the server should parse the **adminId** from the cookie and the **queueId** from the query string. Then, it should query the database for a document that matches those attributes. If a document is found, then the server should respond with a 200 and a response body of { users: [...list of user documents] }. If a document is not found, then the server should respond with 400. If query fails (i.e. Database is down), then the server should return a 500.

*REQ-6: GET /api/admins/queues*

Gets a list of all **queueIds** that the admin manages. The request should have an authentication cookie. The **adminId** should be parsed from the authentication cookie. Then, the server should query the **admins** collection to get a list of queueIds for the specified **adminId**. Then, it should use those queueIds to query the **queues** collection to get a list of **Queue** elements (contains **queueName, description, timeCreated (UNIX timestamp in ms), liveDate + time (UNIX timestamp in ms), closeDate + time (UNIX timestamp in ms), repeatCycle).** The authentication middleware should handle any invalid authentication cookies. If the database queries fail, then the server should respond with a 500.

*REQ-7: GET /api/queue/progress*
This endpoint takes in a request query that contains a **queueId** and a **userId**. If the **queueId** is invalid, then the server returns an error status code and an error message. If the **queueId** is valid, then It finds the **userId** by iterating through the queue. The queue is found by looking through the database. The queried **userId** is compared to the **queue's userId** and stops when a match is found. While the queue is being iterated, there is a counter being incremented that keeps track. If a match is found, the server returns a response body of **userId**, **queueId,** the current place, and the number of people in the queue. If a match is not found, then the server returns an error message saying the user does not exist in the queue. This endpoint essentially finds the user's current position in the queue.

*REQ-8: POST /api/queue/delete*

This endpoint should be used to delete a queue. It takes in a request body containing a **queueId** in a JSON format. The **adminId** should be parsed from the authentication cookie and the cookie should be validated by the middleware. The server should query the MongoDB database for the queue corresponding to the **queueId** and check if the **adminId** that owns the queue matches the parsed **adminId** from the authentication cookie. If it's correct, then the server should proceed to delete the queue document from the database and alert users that a queue has been deleted. If the **adminId** is incorrect, then the server should respond with a 401.

*REQ-9: POST /api/queue/deleteUser*

This endpoint should be used to delete a user in a queue. It takes in a request body containing a **queueId** and an **userId** in a JSON format. The **adminId** should be parsed from the authentication cookie and the cookie should be validated by the middleware. The server should query the MongoDB database for the queue corresponding to the **queueId** and check if the **adminId** that owns the queue matches the parsed **adminId** from the authentication cookie. If it's correct, then the server should proceed to delete the user corresponding to the **userId** from the queue in queue list the database and alerts the deleted user that they have been removed from the queue by the administrator. If the **adminId** is incorrect, then the server should respond with a 401.

## 4.4 Communications Interfaces

All communication between the client and the server will be done through HTTPS 2.0. To prevent malicious access to the server, authentication middleware will be used to prevent non-registered users from accessing **/api/queue/*** and **/api/admins/*** routes. Additionally, each request will be provided with an XSRF token to protect against cross-site request forgery.

HTTP only will be used for development only for simplicity. NGINX will act as a reverse proxy to provide HTTPS functionality in production. If needed, NGINX can also be used for load balancing and rate-limiting to maximize server performance.

# 5. Other Nonfunctional Requirements

## 5.1 Performance Requirements

The website must be interactive and the delays involved must be less. For every action-response of the website, there should be little to no delays. When logging into the admin dashboard, everything from adding admin to database, authentication with OAuth 2.0 using Google or Facebook, getting an access token, sending access token to Google API or Facebook and redirecting admin to dashboard should take less than ~ 2 seconds. In case of logging out, it should take no more than 0.5 seconds for the admin to be redirected to the login page. For the case of administrators, the requirements of creating, deleting, updating, pop, retrieving queues, these should all take ~2 seconds as well in the case of larger queues which might take +1 second to execute after calling these endpoints. For the case of users, joining a queue and displaying their position in the queue should have a latency of less than 1 second.

## 5.2 Safety Requirements

For administrators, requests that pertain to the queue should be securely done over HTTP without affecting the original request. Also the information stored in the queue on users should be only name, phone number and email for point of contact. Information on users should also be safely stored in a database and transmitted over using HTTP.

## 5.3 Security Requirements

For administrators, there should be a proper login mechanism using either Google or Facebook OAuth 2.0. This is to prevent unauthorized users from accessing queue information and stealing customer data. The login system should have a user authentication which checks against the database and makes sures that it is the correct administrator. Additionally, user groups should not be able to see the information of other users but only their own information. The following routes should be protected by checking that a session is valid and active:

1) */api/admins/queues*
2) */api/queue/progress*
3) */api/queues/all*
4) */api/queue/update*

5) *api/queue/pop*
6) */api/queue/join*
7) */api/queue/create*
8) */dashboard/*
9) */api/queue/delete*
10) */api/queue/deleteUser*

## 5.4 Software Quality Attributes

*Correctness and reliability*
The position of the user should always be correct

*Portability*
The waiting line app can be accessed anywhere as long as you have some sort of tablet, phone, computer or laptop.

*Usability*
The website should be very easy for both administrators and users alike to navigate. The features should be very straightforward and obvious as to what to do.

# Appendix A: Work Delegation

| **Team Members** | **Assigned Requirements + Other Duties** |
|---|---|
| Joseph Chen | - **Backend:**<br>    - **4.1.1:** REQ-1, REQ-2, REQ-7<br>- **Frontend:**<br>    - **3.1, 3.2 (UI-1)**<br>- CI/CD + Deployment to AWS EC2 with NGINX |
| Benson Jin | - **Backend**<br>    - **REQ-8** + Automatic login routes + callbacks<br>- **Frontend**<br>    - **Partially 3.2, 3.5** |
| Di Zhou | - **Frontend**<br>    - **3.6, 3.7 (UI-7-UI-16)** |
| Daniel Lee | - **Frontend**<br>    - **3.3, 3.4 (UI-2-6)** |
| Jalen Xu | - **Backend** |

| | |
|---|---|
| | - **4.3.1: REQ-3, REQ-4, REQ-5, REQ-6**<br>- Help with Backend Testing |
| Zhongyuan Tang | - **Backend**<br>  - **4.3.1: REQ-9**<br>  - Help with backend testing with Jest |