Design Doc

Contents

***Important Notes**

- Since I was using my personal email account during the development of this app, upon leaving I have removed my authentication details, so there are no account details in the config file and email notifications will not work. I was also using my personal account to generate service tickets, so this won't work either.
- If anything is unclear in this documentation feel free to reach out ot me at volpestyle@gmail.com.

1.0 Introduction

The purpose of this document is to describe the implementation of NPR's Audio Engineering Discrepancy Sheet. This application is a way to log, view, notify of, and respond to audio engineering discrepancies encountered by NPR employees.

2.0 Design Overview

2.1 Technologies Used

The target platform will be a web application, and the development environment is Visual Studio Code. The app will be built using a M.E.A.N stack (MongoDB, Express JS, Angular, and Node.js). Elastic Search version 6.0.0 will be used to search the database.

2.1.1 Elasticsearch version

It is important that version 6.0.0 is used. This is because when indexing with the module mongoose-elasticsearch-xp, 'es_type' must be set to 'keyword' for all lookups we want to index. Currently, this is only possible with ES v. 6.0.0 or **earlier**.

2.1.2 Startup

It is important that upon first initialization of the app, the technologies are started in this order:

1. Database
2. Elasticsearch
3. Backend
4. Frontend

It is important that the backend is started after ElasticSearch to ensure index mappings are correctly generated.

2.1.3 Config

Inside the 'api/config' folder, there is a default.json file that specifies:

- urls of the frontend, database, and elasticsearch
- api urls used for authentication and getting user information
- twilio and email account info
- json web token settings

Currently, the configuration is:

Angular -> localhost:4200

Express -> localhost:4201

MongoDB -> localhost: 4202

Elasticsearch -> localhost: 4203

2.2 System Architecture

Figure 1 depicts the high-level system architecture.

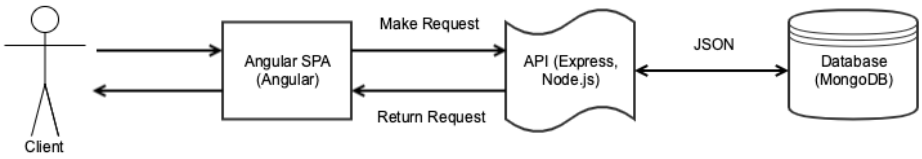


Figure 1: D-Sheet Architecture

2.3 Frontend/Angular

High level components:

- **login** - Makes up the login page. Place for user to enter username and password and attempt login.
- **home -** Landing page.
- **navbar -** At the top of every page with links to all main routes of app.
- **search -** Contains search filters, fields, and search button.
- **results -** Once a search has been made, displays all results.
- **result -** Can display all basic information about a D-sheet. Button to open attachment and buttons to edit/delete. Contains 'entry-notes'.
- **entry-notes -** Displays all notes associated with a D-sheet/interface to add/delete notes.
- **new-entry -** Contains all form elements needed to create a new D-sheet.
- **single-entry-view -** Displays a single D-sheet's information.
- **profile -** Contains all user information. Place to add/delete phone number.
- **subscription -** Displays information about a subscription in readable format. Gives option to delete/remove.
- **shared-sub** - Displays information about a shared subscription in readable format. Primarily used when placed in modal to be selected/added.
- **manage-lookups -** All lookups and lookup values are displayed and made editable by Admin
- **manage-users -** A place to search for all users that are in the NewsFlex api, filtered by the 'isUser' property. Once a user is found, they can be created in the database and edited through the **edit-user-modal**.
- **import-data -** Place to upload .csv containing historic data, and import it into the database.
- **assign-shared-sub -** Place to upload .csv that is a list of names, choose a shared subscription, and apply this subscription to all names in the list.

Entry Components:

- **subscribe-modal -** A modal to create new subscriptions.
- **edit-entry-modal -** A modal to edit an entry.
- **shared-sub-modal -** A modal containing all shared subscriptions. Primarily used to add shared subscriptions to a user, showing which subs the user has and doesn't have.
- **edit-user-modal -** Upon the selection of a user in the **manage-users** component, this modal is opened. If the user exists in the db, the user information will be displayed. If not, one can be created. Once existing, the user's permissions and subscriptions can be edited, and the user can be deleted if needed. Changes here only take effect when 'Apply Changes' is clicked.

Services:

- **auth.service -** Keeps track of users logged in status. Makes api call to check if user is still authorized (token is still valid). Login/Logout
- **auth-guard.service -** Implements 'canActivate()' to block/allow a route for users depending on their authentication status.
- **jwt-interceptor.service -** Upon a 401 unauthorized error, redirects to login page.
- **lookups.service -** Loads in all lookup values from db and sorts them into declared Lookups. Must be provided to any component that uses lookups.
- **data.service -** Handles **all** http requests that access the db.
- **search.service -** Make http request to make an ES search, build an ES query given a RawQuery, output all results returned by query to .csv.
- **roles.service -** Creates a Map containing the status of all permissions for the current user.
- **role-guard.service** - Implements 'canActivate()' to block/allow a route for users depending on their granted permissions.
- **user.service** - Makes request to get token payload
- **user-info.service -** Handles requests that get user information from NewsFlex api v1.
- **Util -** Holds various functions that are unspecific to any one component or service.

Models:

- **Entry -** Frontend representation of an Entry
- **Lookup -** Contains details about a specific lookup**.** Once Lookup Values are loaded in from the database and sorted, they are placed into their respective 'Lookup'.
- **RawQuery -** Data needed to create a query that Elasticsearch can use.
- **SearchSub -** All data associated with a subscription.
- **User -** Frontend representation of a User.

2.4 Backend/API

The backend was made to follow REST guidelines as close as possible.

Routes:

/db - All operations that access database collections.

/user - Accesses the user stored in the current token.

/users - Requests to the NewsFlex Api v1

/search - ElasticSearch operations

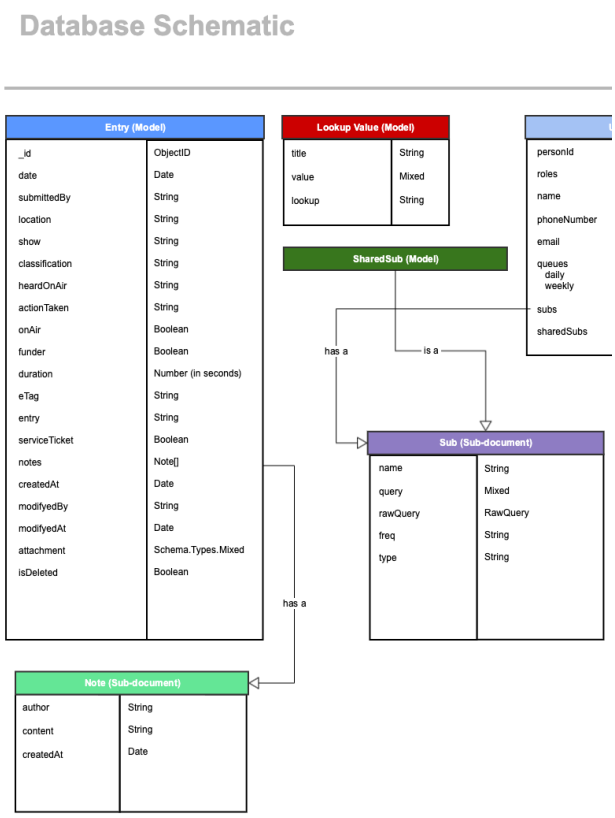
/auth - All operations that maintain the authentication state of the user and create/update the user upon login.

2.5 Database

Technologies used: Mongoose, and Mongo.

Mongoose is an Object Data Modeling (ODM) library, which provides a bit of structure to MongoDB by enabling use of schemas and relations as needed.

Since D-Sheet attachments have a limit of a 100mb (exceeding 16mb), GridFS is the MongoDB specification used to store all file attachments.



Mongoose has a populate feature that allows use of relations. This feature is used:

- Within the QueueItem subdocument, where a given QueueItem's entry references an existing instance of an Entry Model.
- Within the User model, where a user's sharedSubs is an array of references to existing instances of the SharedSub Model.

3.0 User Interface

3.1 Frameworks

Bootstrap 4 and Font Awesome.

3.2 Initial Wireframe

4.0 Implementation Details

Things to Note:

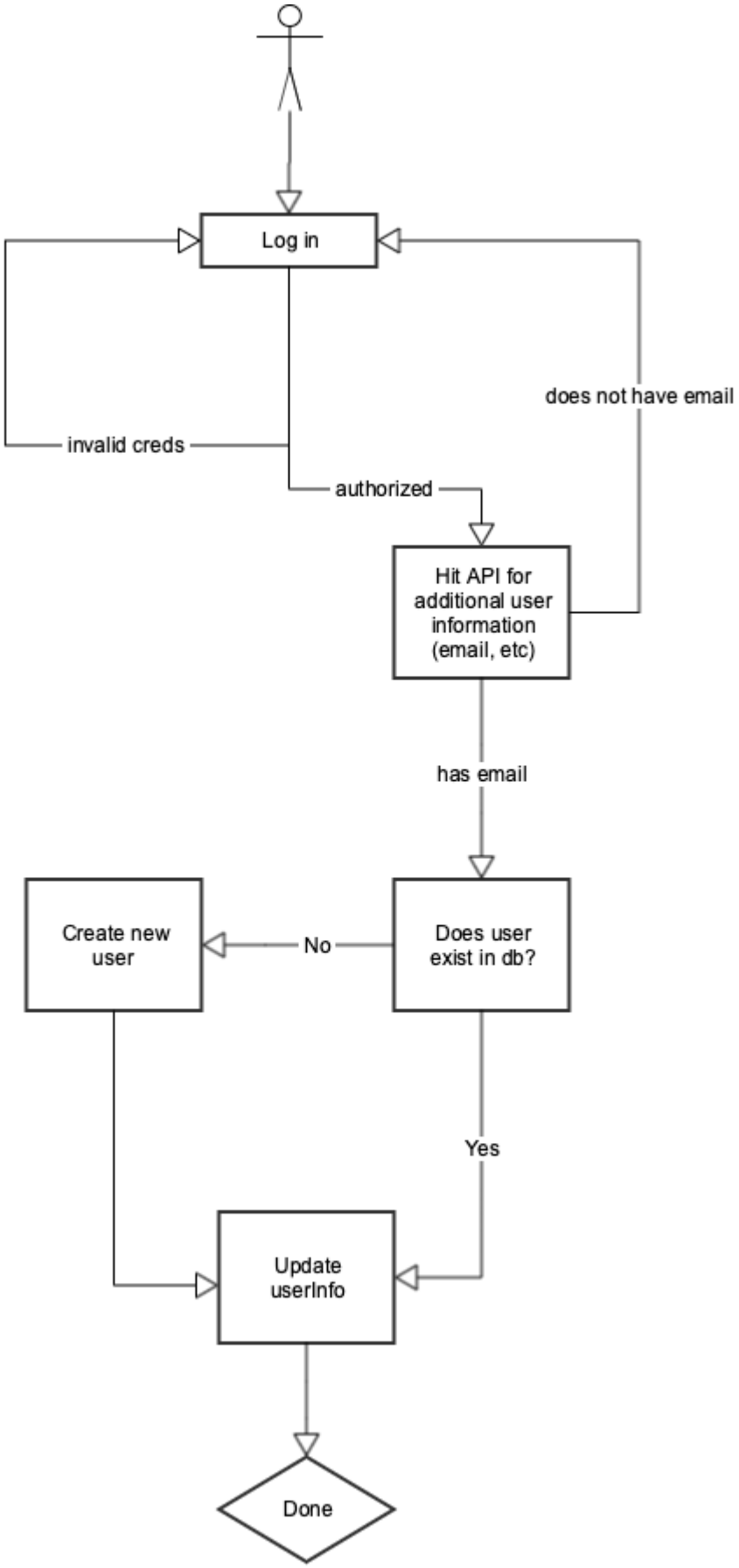
- A user with no email returned by the NewsFlex API cannot log in. However, an Admin can still create an account for them and assign them shared subscriptions. But they will not receive notifications or be able to log in until they have an email. The same goes for text messages.
- If subscription is removed from a user, all items in their queues associated with that subscription will be removed.
- If an Admin deletes a shared subscription, that subscription will be removed from all users that have it.
- Any changes to a users roles/permissions will not take effect until they log in.

4.1 User Authorization

The authentication process is implemented using a local-strategy from Passport and the NewsFlex auth API. A combination of JSON Web Tokens and express cookies is used to keep track of the authentication state.

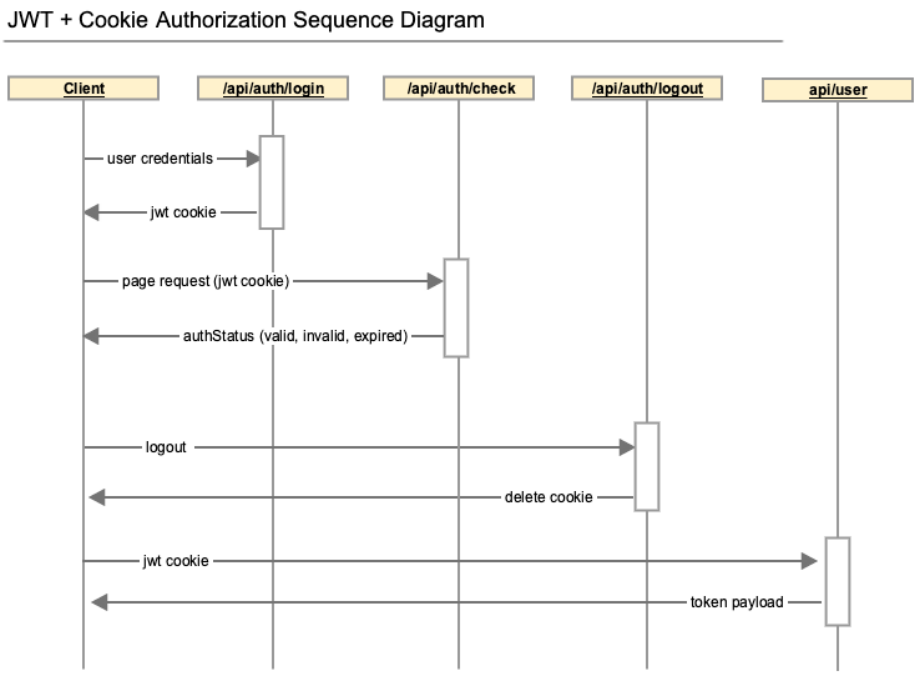
Upon a successful login, the user information returned by the NewsFlex auth API is used to then hit the NewsFlex API v1 to get more information on the user. If the user has no email, they cannot log in**.** If the user exists in the db, their information is updated, if not, a new user is created. This user information is then serialized into a JSON Web Token. This token is then placed inside an HttpOnly cookie, and will be sent with every request made from the client to the backend. The token payload can be accessed by a request to the backend.

It should be noted that the user information inside the token is what the user looked like when they logged in. Any changes to the user during the session will not be reflected inside the cookie. Therefore, when getting user information that may be changed during the user session, an additional call is made to get the updated user from the database. An exception to this is user roles, because while it may change during a session it's not a big deal if the changes are not reflected until the user logs in again.


Log in
invalid creds
does not have email
authorized
Hit API for additional user information (email, etc)
has email
Create new user
No
Does user exist in db?
Yes
Update userInfo
Done

To constantly ensure authentication of the user, an Angular Route Guard is put in front of every route except '/login'. On a page request, this route guard makes a request to the backend to ensure the validity of the current token.

In addition, all backend endpoints that access the database are protected by a jwt-strategy from Passport.



4.2 User Roles/Permissions

Summary

Rather than defining roles and their given actions, the current implementation defines *permissions* and their allowed roles.

All permissions are their allowed roles are declared in 'angular/src/environments/environment.ts'. The 'ADMIN' role has been set as a permitted role for every permission, and there also exists a individual role for each permission. For example, the permission 'manageLookups' is only given to those with the role of 'ADMIN' and 'manageLookups'.

Below is a screenshot of the environment.ts file, showing all the permissions which are restricted to those with at least one of the permitted roles.

```
receiveTexts: { permittedRoles: ['ADMIN', 'receiveTexts'] },
manageLookups: { permittedRoles: ['ADMIN', 'manageLookups'] },
manageUsers: { permittedRoles: ['ADMIN', 'manageUsers'] },
deleteEntry: { permittedRoles: ['ADMIN', 'deleteEntry'] },
editEntry: { permittedRoles: ['ADMIN', 'editEntry'] },
addNote: { permittedRoles: ['ADMIN', 'addNote'] },
addSubToOtherUser: { permittedRoles: ['ADMIN', 'addSubToOtherUser'] },
importHistoricalData: { permittedRoles: ['ADMIN', 'importHistoricalData'] },
canCreateSharedSub: { permittedRoles: ['ADMIN', 'canCreateSharedSub'] },
assignSharedSub: {
  permittedRoles: ['ADMIN', 'assignSharedSub'],
},
```

By default, a user is given no roles, and therefore has none of the declared permissions. Roles can only be given by an Admin or a user with permission to 'manageUsers'.

To restrict certain routes from users without adequate permissions, a second Angular Route Guard is in place where needed.

Details

The permissions of the current user are stored in a Map in the **roles.service.** The map is initialized with a call to getAuthorizations(), which is called on log in and inside every component/service where a users permissions might be needed. However, for efficiency's sake, It does not re-evaluate permissions if the user already has a permissions map.

4.3 D-Sheet Submission

All D-Sheet submissions are handled from the "new-entry" page. Angular Reactive Forms are used to handle all drop down, checkbox, radio, and text input, while more complex input methods, like time and date, are handled with template driven forms and Ng-Bootstrap components. All fields are required before submission with the exception of 'E-Tag' and 'Attachment'. 'Date' and 'Time' will default to the current. The 'submitted by' field is automatically populated by the display name of the current user.

Once a user passes the initial step of filling required fields, a pop-up modal will appear to confirm the submission.

When a D-Sheet is submitted, it is added to the Entry collection and to the queues of any user whose subscribed searches it matches.

4.3.1 File Attachments

When the file attachment field is completed, the 'File' object is saved inside the Entry document, and then the file itself must be sent from Angular to the backend as 'mulitpart/form-data' and stored using GridFS. This is achieved with the help of NPM modules multer and multer-gridfs-storage. Mongoose-gridfs is then used to create a read stream to get the attachments when needed.

4.3.2 D-Sheet Deletion

A D-sheet can be deleted from the search page or a single entry view. Upon deletion, the D-sheet is flagged in the database as 'deleted', but is not actually removed from the database. Once it's flagged, it will no longer appear in search results and cannot be viewed by itself.

4.3.3 Editing a D-Sheet

A D-sheet can be edited from the search page or single entry view. All basic fields can be modified except 'serviceTicket'. Once saved, a 'modifiedBy' and 'modifiedAt' property is added to the entry.

4.3.4 Creating Service Now Tickets

If the 'serviceTicket' property of an entry is set to true upon submission, a service now ticket is created using an XMLHttpRequest. This is handled in 'api /scripts/serviceNow.js'. Currently, only the entry description is being transferred to the ticket.

4.3.5 Importing Historical Data

D-Sheets from the old system can be added into the database as long as it is inside a .csv file, and each row contains the data in this order:

Date - Time - Tech Name - On Air - Show - Entry - Location - Classification - Funder - Duration - What was heard on air - Action Taken

This seems to be the default order exported by FileMaker, and data that is not in this order will not be imported correctly.

4.4 Searching

Elasticsearch is the primary tool used to search the D-Sheets, and with the help of an NPM module, the models created with Mongoose are mapped and indexed into Elasticsearch upon initialization.

When searching, the user can select multiple items from every drop down menu except 'Duration'. With duration, once a selection in made, it can be modified to fit the users needs.

4.4.1 Building a query

There are two ways information for a search query are stored: as a '**raw query**', or a '**query**'. A raw query contains the data for a query in key/value pairs, while a query is the data from a raw query serialized into an object that Elasticsearch can use. This function responsible for the serialization can be found at 'angular/src/app/services/search/search.service.ts'.

Each form of query serve a different purpose, and sometimes both are used. For example, when a user saves a subscription, the raw query is used for displaying the subscriptions criteria, and the query for determining which new entries to send notifications about.

4.4.2 Output to .CSV

Once a search is made, the results can be output to a .csv file. The number of results that can be output is currently 10,000.

4.5 Lookups

All lookup values are stored in a single collection in Mongo, each with a name tag determining the lookup it belongs to. When needed, a service will load and sort them according to their 'lookup' property.

Below is an example of how they would be loaded in directly from the database.


```
(8) [{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}]                    lookups.service.ts:25
  0: {_id: "5d165bb24453b72de3c22aec", val: "All things considered", lookup: "show", __v: 0}
  1: {_id: "5d165c474453b72de3c22aee", val: "NewsFlex", lookup: "classification", __v: 0}
  2: {_id: "5d165c634453b72de3c22af0", val: {…}, lookup: "duration", title: "0 +", __v: 0}
  3: {_id: "5d165c784453b72de3c22af1", val: {…}, lookup: "duration", title: "30s to 1h", __v: 0}
  4: {_id: "5d165c844453b72de3c22af2", val: "Start/Stop Error", lookup: "actionTaken", __v: 0}
  5: {_id: "5d165c8c4453b72de3c22af3", val: "Train Wreck", lookup: "heardOnAir", __v: 0}
  6: {_id: "5d165ff84453b72de3c22af4", val: "Booth 6", lookup: "location", __v: 0}
  7: {_id: "5d1a1a8de59ba9158251fc79", val: "Remote", lookup: "location", __v: 0}
  length: 8
  __proto__: Array(0)
```


... and then sorted into a usable variable.


```
                                                                 lookups.service.ts:27
{show: Array(1), classification: Array(1), duration: Array(2), actionTaken: Array(1), heardOnAir: Array(1), …}
  actionTaken: Array(1)
    0: {_id: "5d165c844453b72de3c22af2", val: "Start/Stop Error", lookup: "actionTaken", __v: 0}
    length: 1
    __proto__: Array(0)
  classification: [{…}]
  duration: (2) [{…}, {…}]
  heardOnAir: [{…}]
  location: (2) [{…}, {…}]
  show: [{…}]
  __proto__: Object
```


The lookups themselves are defined in the service, which allows for easy removal/addition of lookup categories.

4.6 Personal Search Subscriptions

From the search page, the user has the option to subscribe to the current search. By subscribing to a search, the user opts in to receive notifications whenever a D-sheet is created that would be contained in the results of the subscribed search. This saves the current query to the user's list of subscriptions in Mongo. Each subscription has a name (inputted by the user), type (email or text), and frequency (immediate, daily, weekly, etc). If a subscription type is 'text', the only frequency option available is immediate.

The user must have a phone number saved with their profile before saving a text subscription. If the use tries to delete their phone number while they have text subscriptions, they will be prompted to delete them first.

4.6.1 Notification Types

Email: HTML body containing the name of the subscription that picked up the new entry, all basic entry information, and a link to view the entry online.

Text: A text body containing the name of the subscription that picked up the new entry and a link to view the entry online.

In order for a user to create a text subscription, they must have a phone number associated with their account.

4.6.2 Notification Frequencies

Immediate: The notification is send to the user instantly. If a user has multiple subscriptions that pick up the same entry, a single notification will still be sent.

Digest (Daily, Weekly, etc): A single notification containing all subscriptions of a given frequency is sent to the user. There may be duplicate entries if more than one subscription picks up a given entry.

The user has a 'queue' for all digest frequencies. As new D-sheets are created, the queues are filled accordingly and emptied periodically with the help of node-scheduler.

4.6.2 Sending Notifications

Sending Emails

All emails are sent using NodeMailer. There details of the account to send emails from is located inside the config file.

Scheduling Digest Emails

With the help of node-scheduler, the daily and weekly queues of every user are emptied at 5:00 PM every day and 5:00 PM of every Friday respectively.

Sending Texts

All texts are sent using Twilio.

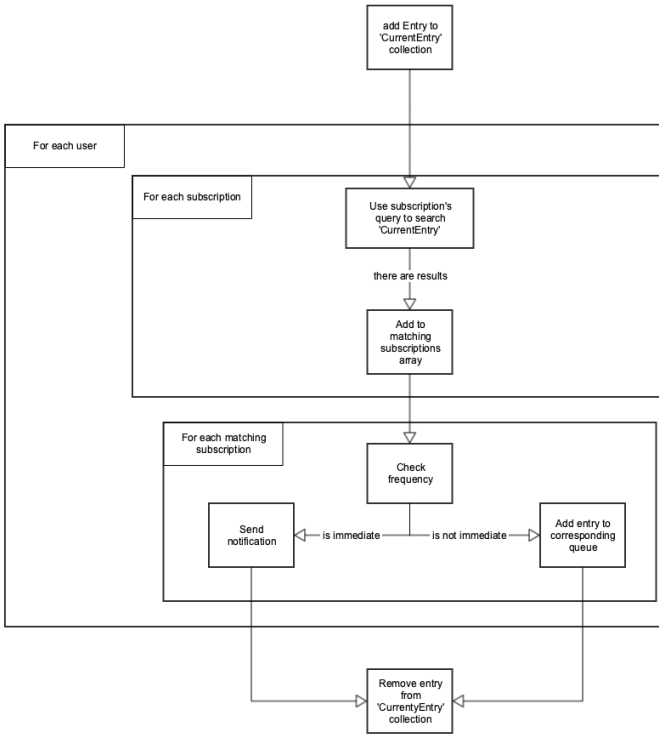
4.6.1 Handling New D-Sheets

Any time a new D-sheet is created, all subscriptions who's criteria match that of the new D-Sheet must be found in order to appropriately handle sending notifications. The user should also know which subscription triggers each notification they get.

The basic idea is to iterate through all users and their subscriptions (personal and shared), getting the subscriptions whose query would return the new entry. For each of the subscriptions whose frequencies are not 'immediate', it's name and the entry Id are placed into the user's corresponding queue to be sent later. The subscriptions with immediate frequency are sent out on the spot. However, if a user has multiple immediate frequency subscriptions that match the new entry, we don't want the user to receive redundant immediate notifications about the same entry, so it is described to the user that this D-Sheet has been picked up by multiple subscriptions.

4.6.1.1 The 'CurrentEntry' Model

So how do we determine which subscriptions will send out the new entry? Since ElasticSearch can only search collections, one way is to use a subscription's query to search the entire collection, and if the new entry appears in the results, then the subscription should send out a notification. However, this requires searching the entire collection for every subscription, which doesn't seem very efficient. So instead, I created a new model called 'CurrentEntry', identical to the Entry model. This model will have its own collection for Elastic Search to search through. This way, when a new D-sheet is created, it is added to the collection, and ElasticSearch just has to search through a single entry. Therefore, if there are any results returned at all by the search, we know the subscription should sent out a notification. However, it is critical that immediately after all users and all subscriptions are handled, the collection is emptied.



4.7 Shared Search Subscriptions

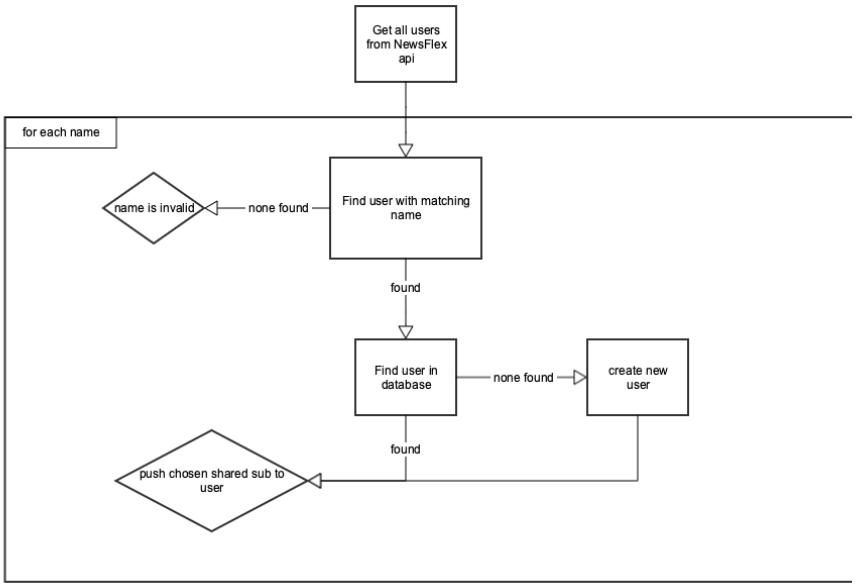
As an object, shared subscriptions differ from personal subscriptions by the 'author' property, where a shared sub has an author and a personal subscription does not. Otherwise, they look and function exactly the same.

A shared subscription can be created from the search page, only by those with the permitted roles.

Any user can subscribe to a shared subscription from their profile page, where they will be prompted with a list of all shared subscriptions.

4.7.1 Mass Assigning of Shared Subscriptions

Shared subscriptions can be assigned to many users at once from the 'assign-shared-sub' component. The users are determined using a .csv of comma separated first and last names. Once the list has been processed, a summary modal will appear listing all the names that weren't valid users, already had the subscription, and don't currently have an email. Here is the basic process:


Get all users from NewsFlex api
for each name
Find user with matching name
none found
name is invalid
found
Find user in database
none found
create new user
found
push chosen shared sub to user