New York Institute of Technology

Senior Design Project Report

CSCI 455 - M01/ Spring 2021

Mentor: Dr. Li

Project Name: Scholar Seek

Team Name:

Group Member:

Hui (Henry) Chen	hchen60@nyit.edu	1242445
Jungi Park	jpark23@nyit.edu	1259364
Michael Trzaskoma	**mtrzasko@nyit.edu**	**1202901**
Gregory Salvesen	gsalvese@nyit.edu	1256999
Zakaria M Khan	zkhan15@nyit.edy	1213675

Table of Contents

Introduction

Abstract

Scholar Seek is a continuation of an existing CSCI426 (Information Retrieval) project, of which team members Hui Chen and Michael Trzaskoma helped develop. Scholar Seek is a cross platform mobile application (compatible with IOS and Android) that serves as a “one stop shop” for prospective college students in discovering their academic major, college, and relevant scholarships. Scholar Seek aims to provide these services accurately while respecting and preserving the privacy and security of the user. Users can manually browse scholarships, colleges, and majors by selecting relevant categories. Users can also receive a custom recommendation for scholarships, colleges, and major categories based on survey input. Users may also bookmark individual listings of their choice and view their recent browsing history.

The previous project, “Scholarship Recommendation App”, was a mobile application that provided manual scholarship browsing by category and custom scholarship recommendations for a user based on a short registration and an optional survey. The continuation of this project, Scholar Seek, expands the features of the existing app while implementing new, more robust features including an enhanced database, distinguishing this project from its previous state. The new planned features will also expand the scope of the app. The project, in its state from CSCI426, serves as a scholarship recommendation service, using a user’s profile information to match them as best as possible with scholarships from scholarships.com. Scholar Seek expands this and improves this feature by assisting prospective college students find the higher education institution that best suits them in addition to discovering potential majors. Our

CSCI455-Project will help students who are financially struggling find the best scholarship, explore potential majors, and find the college best suited for them by using their profile information to recommend likely candidates.

Motivation

Attending college comes with a heavy price tag that in many cases cannot be paid in full upfront. Even with a college savings plan or aid from FAFSA (Free Application for Federal Student Aid), paying for college can be a tremendous burden that impacts students for the rest of their lives. Historically the price of higher education is only increasing. Even when adjusted for inflation the increase in cost for attending higher education is increasing substantially. According to the National Center for Education Statistics, just between 2007 and 2017 there was a 30 percent increase in the cost of higher education at public institutions. Additionally with the world recovering from the corona virus pandemic, many have struggled even more to pay for higher education. Thus financial aid is more important than ever now - particularly scholarships.

Furthermore the decision to undertake a significant financial burden to pursue higher education is not one to be taken lightly. Students must be well prepared to understand what academic and professional realities await them following their decision. As college students, the Scholar Seek Team is personally aware of the difficult process of discovering a suitable area of study, a college of best fit, and determining an acceptable cost. As students that had to go through the decision making process to figure out where to go and for what major, and for how much, it’s a very overwhelming experience. In many cases, such as in inner city high schools, students receive little guidance from educators in making their college decisions. Many are pushed into deciding on higher education without coming to terms with the fiscal or academic realities of the decision. Scholar Seek aims to assist students in discovering their academic interests, in understanding the professional realities that might accompany them, while also assisting them in financing their education and in searching for a higher education institution of best fit. Scholar Seek aims to accomplish these goals through its ability to provide scholarships, colleges, and majors to users both through manual browsing and a recommendation system based on user survey input.

Scope

Scholar Seek is intended to assist primarily high school students in navigating their college search process. However the app is not limited to this market audience and can still be used by undergraduate students seeking scholarships or a transfer. Scholar Seek is intended to serve students who are interested in attending higher education within the United States. While Scholar Seek does provide custom recommendations, it does not claim to provide the most perfectly catered selection for the user, nor does it claim to do all the research for the user. It does however aim to provide the user with a guided baseline from which they can expand their own research or from where they can narrow their focus. This is especially pertinent in the major recommendation in which a field or category of majors is suggested to the user for further exploration and discovery.

Related Works

Scholly, perhaps the most prestigious scholarship app on the market, featured on shark tank and recommended by many teachers nationwide is the most popular scholarship recommendation app. It is a subscription service where users create profiles based on their interests, achievements, and unique characteristics. Examples of questions asked during profile creation include gender, race, GPA, state, and grade levels. When the user profile is complete, Scholly uses an embedded recommendation algorithm to match the scholarships that best fit the profile and create a priority list. The recommendation may be searched by deadline or scholarship amount. Each listed scholarship has a link to the application website, which can be stored in a saved personal list, which allows users to add an application deadline to your dashboard calendar [1]. It should be noted that while offering some free services, scholly is quite heavily monetized which for many students who depend on scholarships for the financial viability of their education, is an insulting and degrading prospect.

Scholarships.com is a scholarship database website that allows users to search or navigate manually. The website includes 3.7 million scholarships, totaling 19 billion U.S. dollars. Users can also create profiles to narrow or filter scholarships according to their preferences. When users manually search for scholarships in a directory, the available scholarships are organized by a wide range of categories that include subcategories of scholarships listed [2].

The College Board's Scholarship Search engine has been an industry leader in scholarships and grants. The College Board is affiliated with more than 2,200 programs that support scholarships and grants. The search engine system has an algorithm where the user enters personal and academic information and displays the scholarship to which he or she is qualified. The personal information required includes gender, citizenship, minority background, disability, blindness, parent/spouse thinking, racial background, religion, and special conditions. There is also a complete page for personal information to be filled out for a particular organization, such as an employer, club/organization, or military status. The required academic information includes degree levels, educational status, and a major. The College Board scholarship search also filters out what types of awards students can receive, allowing users to find help based on academic performance or proven financial needs [3].

Niche is one of the sites that provide college and school liable information to students and family. Niche's discovery platform analyzes public data sets and reviews to fabricate reliable rankings, and report cards and profiles for every k-12 school, college and neighborhood in the United States. [4] They provide 140 million reviews and ratings on every school and college in America. Niche helps many students and families find and enroll in the best fit school for them. Also, they provide various categories, such as career and major. Niche's platform, data, and services has helped thousands of students and school recruiters. [5]

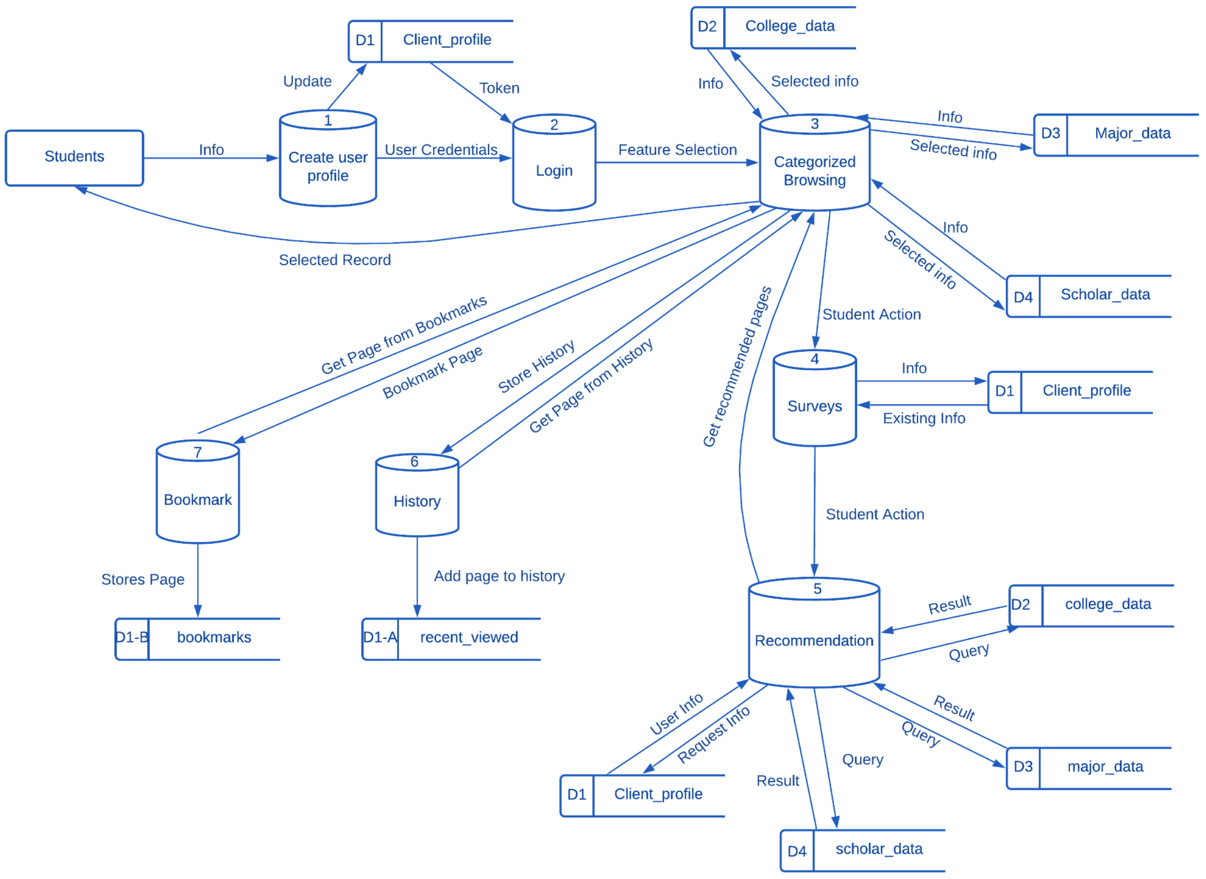
Scholarship Recommendation was a CSCI 426 project - Information Retrieval. The project mainly provided a scholarship recommendation that had taken factors, such as gender, age, areas of residence, SAT scores, and ethnicity. The recommendation model takes the factors, and the user acquired relevant best fit scholarship from the model. Also, users could brower other various scholarships by categorized sections, such as academic major, GPA, Age, artistic ability, and athletic ability, which allowed users to navigate to most applicable scholarship information. [6]

Analysis of the System

Activity List

- Scholarships: relevant category & custom recommendation
- Colleges: relevant category & custom recommendation
- Majors: relevant category & custom recommendation
- Bookmark scholarships, colleges, and majors for viewing
- History: view scholarships, colleges, and majors that have been browsed

Data Flow Diagram



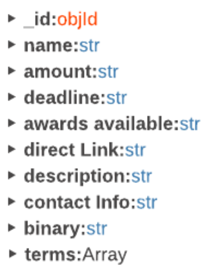
(Figure 2.1-1, a snapshot of Data Flow Diagram for Scholar Seek)

Database Design

The entire project utilizes MongoDB as our first preference. MongoDB was selected for its open-source NoSQL database, rich community support, and availability locally. MongoDB does not have the read and writes (I/O) limitation compared to Google Firestore as we are aware from the existing CSCI 426 project that the scholarship web scraping program requires 11 reads (input) and 7 writes (output). Also, MongoDB has a number of flexible APIs for our backend when querying a particular collection, document, or attribute.

Implemented Database Design

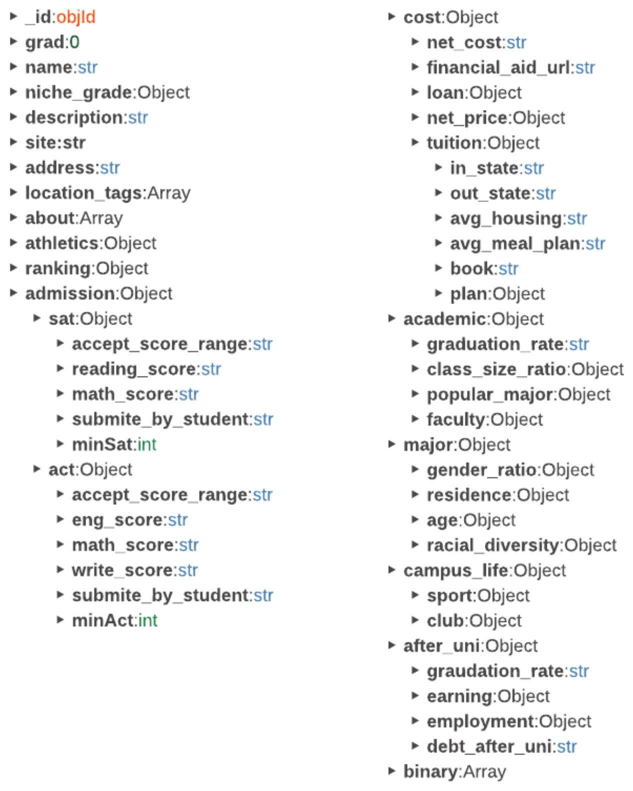
There are four major databases throughout the entire project.



```
▸ _id:objId
▸ name:str
▸ amount:str
▸ deadline:str
▸ awards available:str
▸ direct Link:str
▸ description:str
▸ contact Info:str
▸ binary:str
▸ terms:Array
```

(Figure 3.1-1, the Scholarship data structure)

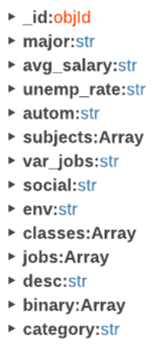
The scholarship document is utilized to store all scholarship information while we scrape the data from scholarships.com. The unique identifier of the document is the “_id” and “name” attributes. The “binary” attribute is what we digitized the characteristics of the scholarship during the web scraping. Digitizing the characteristics of the scholarship allows our scholarship recommendation model to compare the user's profile against the scholarship properties. Then the “term” attribute is the unique term about this scholarship.


```
▸ _id:objId                          ▸ cost:Object
▸ grad:0                                 ▸ net_cost:str
▸ name:str                               ▸ financial_aid_url:str
▸ niche_grade:Object                     ▸ loan:Object
▸ description:str                        ▸ net_price:Object
▸ site:str                               ▸ tuition:Object
▸ address:str                                ▸ in_state:str
▸ location_tags:Array                        ▸ out_state:str
▸ about:Array                                ▸ avg_housing:str
▸ athletics:Object                           ▸ avg_meal_plan:str
▸ ranking:Object                             ▸ book:str
▸ admission:Object                           ▸ plan:Object
    ▸ sat:Object                     ▸ academic:Object
        ▸ accept_score_range:str         ▸ graduation_rate:str
        ▸ reading_score:str              ▸ class_size_ratio:Object
        ▸ math_score:str                 ▸ popular_major:Object
        ▸ submite_by_student:str         ▸ faculty:Object
        ▸ minSat:int                 ▸ major:Object
    ▸ act:Object                         ▸ gender_ratio:Object
        ▸ accept_score_range:str         ▸ residence:Object
        ▸ eng_score:str                  ▸ age:Object
        ▸ math_score:str                 ▸ racial_diversity:Object
        ▸ write_score:str            ▸ campus_life:Object
        ▸ submite_by_student:str         ▸ sport:Object
        ▸ minAct:int                     ▸ club:Object
                                     ▸ after_uni:Object
                                         ▸ graudation_rate:str
                                         ▸ earning:Object
                                         ▸ employment:Object
                                         ▸ debt_after_uni:str
                                     ▸ binary:Array
```

(Figure 3.1-2, a snapshot of college data structure)

The college document is utilized to store all college information that we scraped from niche.com. The unique identifier of the document is the “_id” object and “name” of the college. We aggregated some of the information as a sub-attribute. For instance, we have an admission attribute, which consists of admission requirements, deadline, direct application link, and etc. In that way, we are able to quickly locate a particular attribute when our application queries the database by providing the correct key term(s). Then the “binary” attribute is what we have designed to digitize the characteristics of the college.

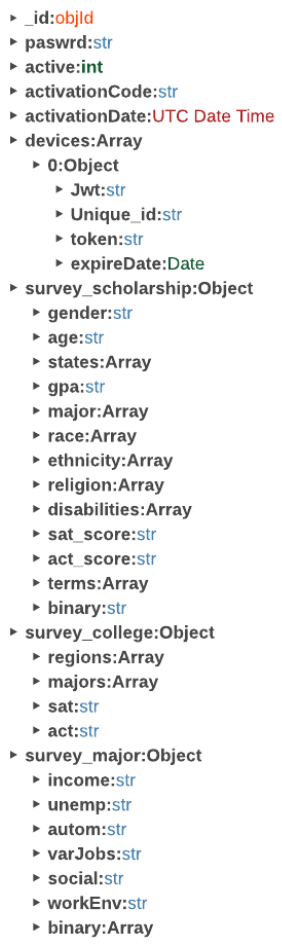
In that way, we are able to compare the user's college preference profile with college characteristics.



```
▸ _id:objId
▸ major:str
▸ avg_salary:str
▸ unemp_rate:str
▸ autom:str
▸ subjects:Array
▸ var_jobs:str
▸ social:str
▸ env:str
▸ classes:Array
▸ jobs:Array
▸ desc:str
▸ binary:Array
▸ category:str
```

(Figure 3.1-3, a snapshot of Major data structure)

The major document has two unique identifiers which are the "_id" and "major". We aggregated some attributes as sub-attributes. For example, a major may have multiple potential classes that each student can take. Then, the binary attribute is digitized of the major's characteristics. Once again, we utilized the "binary" attribute to compare user's major preferences in order to recommend majors to the users. The "category" attribute is what we have merged from scholarships.com and niche.com categorized this major.



```
▸ _id:objId
▸ paswrd:str
▸ active:int
▸ activationCode:str
▸ activationDate:UTC Date Time
▸ devices:Array
    ▸ 0:Object
        ▸ Jwt:str
        ▸ Unique_id:str
        ▸ token:str
        ▸ expireDate:Date
▸ survey_scholarship:Object
    ▸ gender:str
    ▸ age:str
    ▸ states:Array
    ▸ gpa:str
    ▸ major:Array
    ▸ race:Array
    ▸ ethnicity:Array
    ▸ religion:Array
    ▸ disabilities:Array
    ▸ sat_score:str
    ▸ act_score:str
    ▸ terms:Array
    ▸ binary:str
▸ survey_college:Object
    ▸ regions:Array
    ▸ majors:Array
    ▸ sat:str
    ▸ act:str
▸ survey_major:Object
    ▸ income:str
    ▸ unemp:str
    ▸ autom:str
    ▸ varJobs:str
    ▸ social:str
    ▸ workEnv:str
    ▸ binary:Array
```

(Figure 3.1-4, a snapshot of the client's profile)

The client's profile is where we stored everything that is relevant to the client. In this case, we stored the user's devices as an array, and each survey results to its respective attributes by aggregating them. Under each survey, the binary attribute is populated when the clients try to capture the survey answer. In that way, we could digitize the user's respective profile for each recommendation.

<u>Functionality and Implementation</u>

______The project was designed using React Native and Javascript for the front end and a combination of technologies for the backend including Selenium and Python for scripting, AWS for hosting, and MongoDB for storage, along with Flask to build RESTful API and connect React Native to the backend. Docker was used to encapsulating and deploying our entire system. Json Web Tokens (JWT) were also used to encrypt the communication between the frontend and the backend. Postman was used to testing the APIs. Lastly, the Ngrok was utilized to create a tunnel network for the test environment.

React native was chosen due to its cross-platform functionality, allowing development for Android, iOS and web simultaneously, as well as prior experience from team members. Python was chosen to run the backend because of a strong Python foundation from backend developers, as well as its numerous tools designed for web scraping, which was crucial for the success of this project. Python also works well with MongoDB using PyMongo, and with Selenium for testing making it the ideal choice for the project.

<u>Features</u>

The team was able to implement almost all planned features. These included authentication, manual browsing and recommendations for scholarships, majors, and colleges, and the bookmarking and recent viewing of specific listings.

The authentication system from the previous project, which utilized Google services API to handle sign up and login, was replaced by an in-house secure token based registration and authentication system. The benefits for the user include improved security but also a more convenient authentication experience. With the new registration and login, users are not required to fill out any survey and do not have to provide any personally identifiable information other than an email address. After a seamless verification email response, the user can login and begin to use the app.

The original core feature of the app, the scholarship recommendation, was improved from its initial iteration. Scholar Seek provides relevant scholarship recommendations based on several factors. Some are required in order to generate a successful recommendation while others are optional but can help improve the relevance of the results. The mandatory factors are gender, age, state of residence or schooling, and GPA. Optional factors include SAT and ACT scores, academic major, race, religion, disabilities (if any), and ethnicity. Multiple options can be selected for the following fields: state of residence or schooling, academic major, race, religion, disabilities, and ethnicity. The ability to select multiple fields is a marked improvement from the previous state of the project, opening the app to be more accessible to a more diverse user base. The user experience during survey submission was also improved as state of residence or schooling and many of the optional fields can now be selected through a searchable drop down multiple select menu rather than manually typed by the user as in the previous project. The surveys are now also editable and some data is even shared between surveys such as with the college and scholarship surveys. Additionally, the privacy of the app was enhanced as date of birth and zip code fields from the previous project were replaced with age and state of residence and or schooling which are much less sensitive and much less identifying while still being sufficient to deliver a relevant recommendation.Upon submitting a scholarship survey, the user can receive a custom recommendation of scholarships. The user is given a table containing the recommended scholarship listings, where they can sort by title, scholarship amount, and deadline. The user can also manually browse scholarships by selecting relevant categories such as academic major, gpa, age, state, deadline, employer, ethnicity, financial need, gender, ACT/SAT score, race, religion, military affiliation, organization, and many others. The user can access manual browsing without submitting any survey data.

Scholar Seek now offers a college browsing and recommendation service, a feature that was absent from the previous iteration of the project. Colleges can be manually browsed by state and a custom recommendation can be received by submitting a college survey. The college survey requests the user to enter their preferred college location (selected through a searchable drop down menu from which multiple options can be chosen), their SAT/ACT scores, and their major of interest (as some colleges are better known for some programs as opposed to others). The user is required to submit two out of three fields of their choice. The user can choose to submit either the exam scores (of which only SAT or ACT is required) and a location, or the exam field and a major preference or alternatively, state and major preferences. If the user has previously submitted a scholar survey, their region preference and their major preference (if they submitted one) will be present.

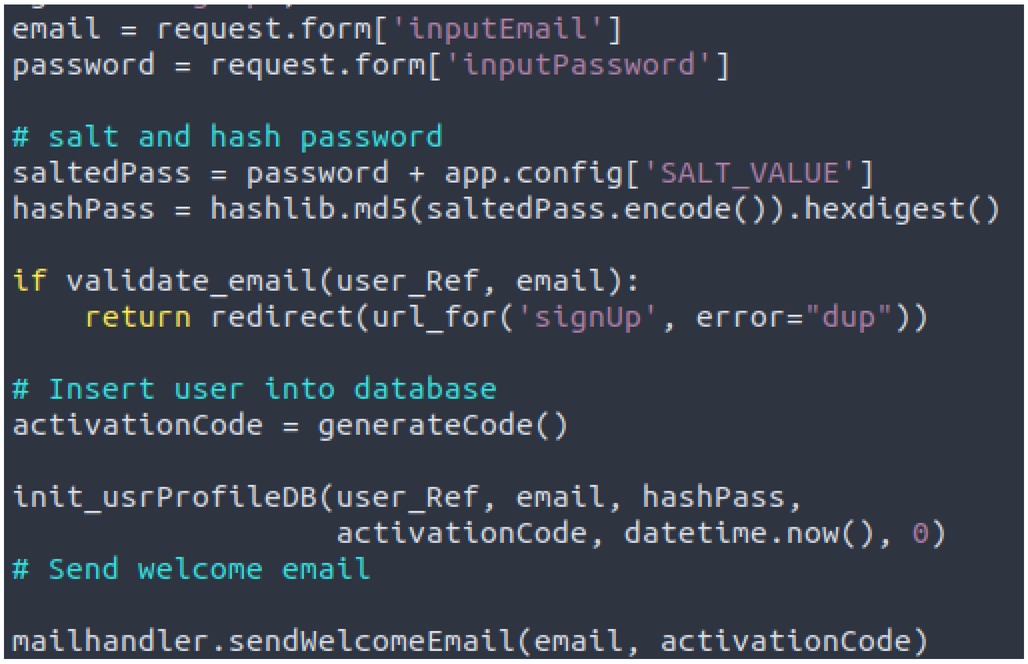
Scholar Seek now also provides manual browsing for academic majors and a recommendation service to recommend a general area of majors to users (for their further exploration). Scholar Seek's outcomes based approach for recommending academic fields of study is designed to encourage students to consider important factors often neglected by other tools and guides such as common required coursework, expected salary, and other professional lifestyle realities including social interaction, variety of jobs in the industry and other such factors.

The listings viewed by users will be added to their browsing history which can be viewed from the main feature pages (relevant to the respective categories) or from the account screen. Up to fifteen listings can be stored in the user's history. This feature was also not present in the prior iteration of the project. Listings can be sorted by title and type.

The user can now also bookmark listings. The user can accomplish this either via a long press from the list view via a pop up modal or from a toggle-able button at the top of the detail page for each listing. Bookmarks can also be unbookmarked from the detail page as well as via long press from the view bookmarks page. Bookmarks can be sorted by title and type.

Account Registration & Verification

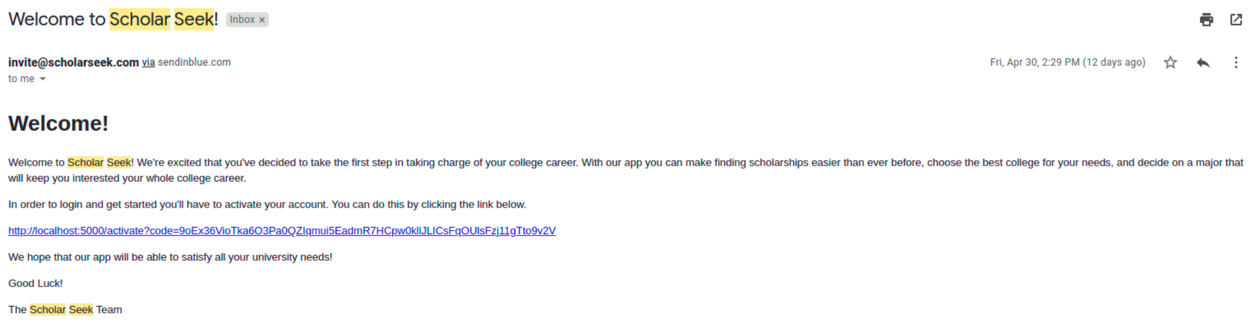
There is two parts in this feature that ScholarSeek could do:



```
email = request.form['inputEmail']
password = request.form['inputPassword']

# salt and hash password
saltedPass = password + app.config['SALT_VALUE']
hashPass = hashlib.md5(saltedPass.encode()).hexdigest()

if validate_email(user_Ref, email):
    return redirect(url_for('signUp', error="dup"))

# Insert user into database
activationCode = generateCode()

init_usrProfileDB(user_Ref, email, hashPass,
                  activationCode, datetime.now(), 0)
# Send welcome email

mailhandler.sendWelcomeEmail(email, activationCode)
```

(Figure 4.2-1, Screenshot of the sign up code)

When a user signs up their email and password are retrieved from the HTML forms. The password is then salted with a secret key, and hashed using the MD5 algorithm. If the email already exists in the database, a duplicate error is returned to the user. Otherwise, an activation code is generated, the new user is inserted into the database and a welcome email is sent using the email provided, which will prompt them to activate their account using a link provided.



Welcome to Scholar Seek! Inbox ×

invite@scholarseek.com via sendinblue.com
to me

Fri, Apr 30, 2:29 PM (12 days ago)

Welcome!

Welcome to Scholar Seek! We're excited that you've decided to take the first step in taking charge of your college career. With our app you can make finding scholarships easier than ever before, choose the best college for your needs, and decide on a major that will keep you interested your whole college career.

In order to login and get started you'll have to activate your account. You can do this by clicking the link below.

http://localhost:5000/activate?code=9oEx36VioTka6O3Pa0QZIqmui5EadmR7HCpw0klJLICsFqOUlsFzj11gTto9v2V

We hope that our app will be able to satisfy all your university needs!

Good Luck!

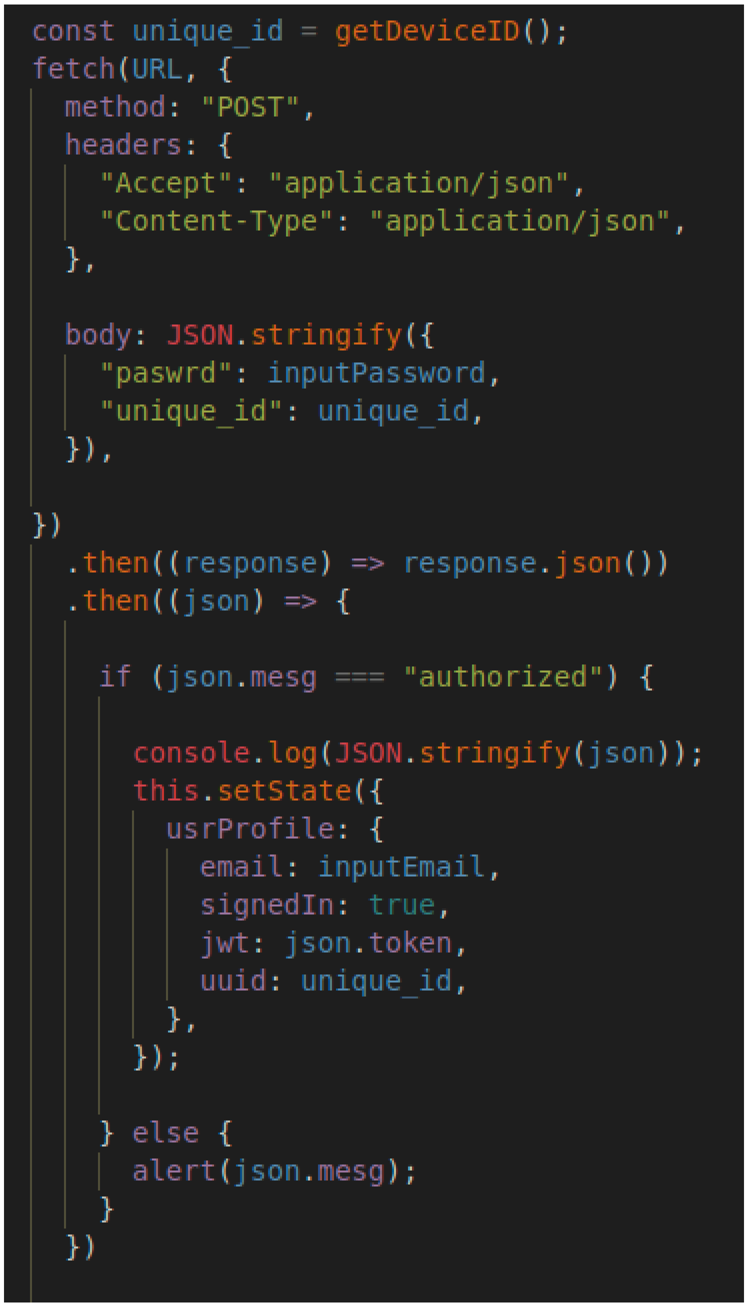
The Scholar Seek Team

(Figure 4.2-2, Screenshot of the welcome email)

This is a screenshot of the welcome email that a new user will receive when signing up for an account.

<u>Token verification</u>

ScholarSeek enables the user to have multiple devices under one account. Each device will assign a UUID code which will be sent to the backend.

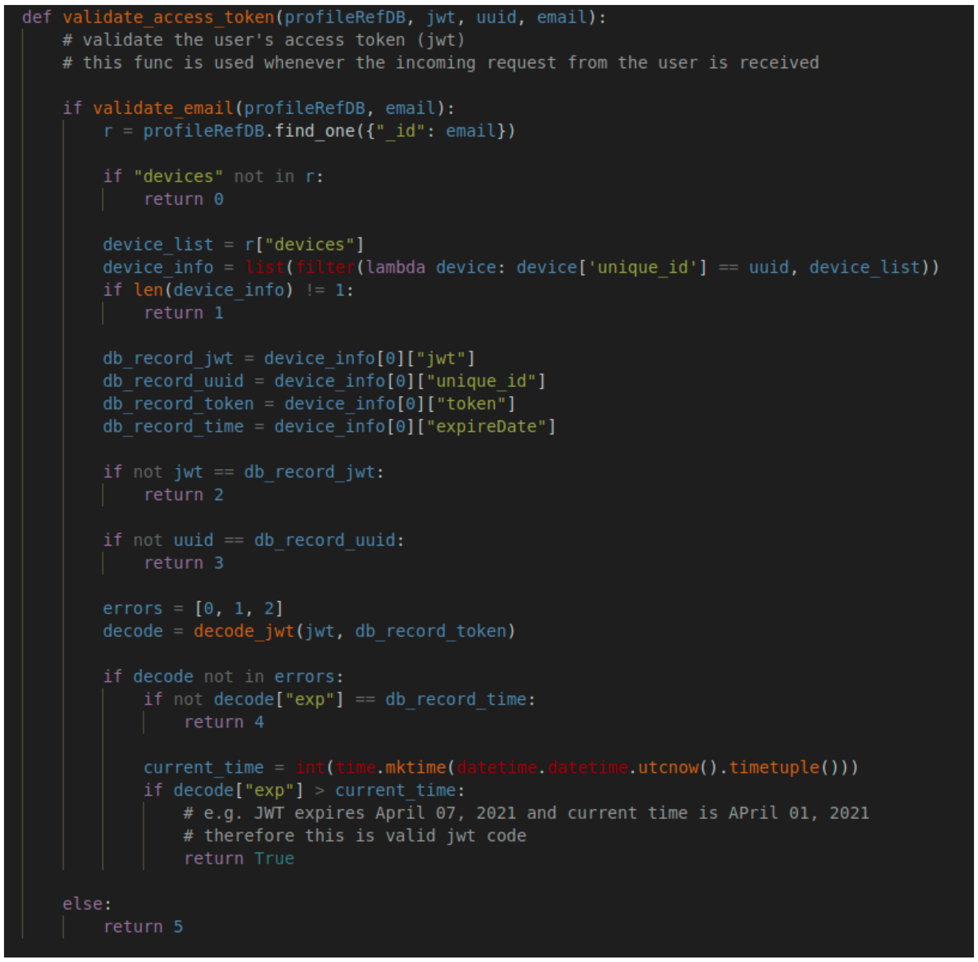


```
const unique_id = getDeviceID();
fetch(URL, {
  method: "POST",
  headers: {
    "Accept": "application/json",
    "Content-Type": "application/json",
  },

  body: JSON.stringify({
    "paswrd": inputPassword,
    "unique_id": unique_id,
  }),

})
  .then((response) => response.json())
  .then((json) => {

    if (json.mesg === "authorized") {

      console.log(JSON.stringify(json));
      this.setState({
        usrProfile: {
          email: inputEmail,
          signedIn: true,
          jwt: json.token,
          uuid: unique_id,
        },
      });

    } else {
      alert(json.mesg);
    }
  })
```

(Fig 4.3-1, a snapshot of how the frontend sends an API call and get authenticated)

The unique device ID is generated on the client's side through Expo API. Once the unique ID is generated, it will make an API call via POST method. Inside the post method, the user's password and unique device ID will be sent as a body message. Then the API will respond to the call by returning a JWT code if the user successfully enters the correct credentials. An alert will notify the user if the incorrect credentials are inputted.

For every http method (GET, POST, PATCH, and DELETE) that we utilized throughout the project, we required three basic information from the user: email address, unique device ID, and JWT code. These four pieces of information are essential to identify the user role and their identity as we do not want the unauthorized user to access any of the resources on our backend or database.



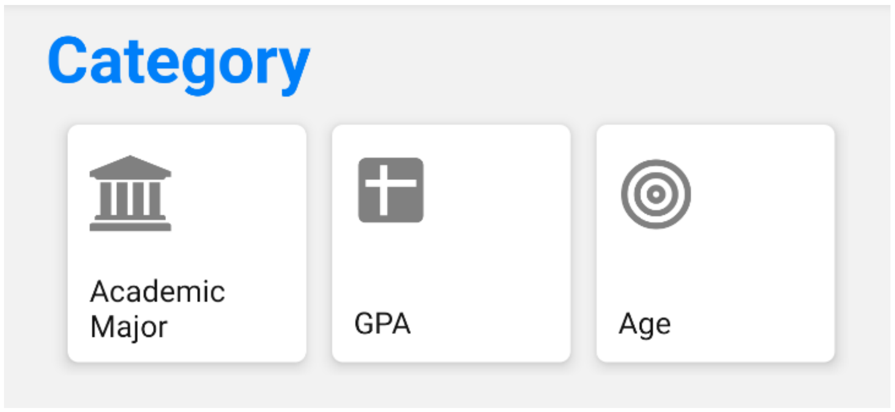
```python
def validate_access_token(profileRefDB, jwt, uuid, email):
    # validate the user's access token (jwt)
    # this func is used whenever the incoming request from the user is received

    if validate_email(profileRefDB, email):
        r = profileRefDB.find_one({"_id": email})

        if "devices" not in r:
            return 0

        device_list = r["devices"]
        device_info = list(filter(lambda device: device['unique_id'] == uuid, device_list))
        if len(device_info) != 1:
            return 1

        db_record_jwt = device_info[0]["jwt"]
        db_record_uuid = device_info[0]["unique_id"]
        db_record_token = device_info[0]["token"]
        db_record_time = device_info[0]["expireDate"]

        if not jwt == db_record_jwt:
            return 2

        if not uuid == db_record_uuid:
            return 3

        errors = [0, 1, 2]
        decode = decode_jwt(jwt, db_record_token)

        if decode not in errors:
            if not decode["exp"] == db_record_time:
                return 4

            current_time = int(time.mktime(datetime.datetime.utcnow().timetuple()))
            if decode["exp"] > current_time:
                # e.g. JWT expires April 07, 2021 and current time is APril 01, 2021
                # therefore this is valid jwt code
                return True

    else:
        return 5
```

(Fig 4.3-2, a snapshot of how the backend would validate the JWT)

Whenever there is an incoming HTTP request, JWT will be checked. The first checkpoint is to check does this email exist on the database. Then the second checkpoint is if there is any matched JWT under the client's profile. The third checkpoint is using a JWT associated token to decrypt in order to compare the unique device ID and JWT expiration timestamp under the client's profile. If the user does pass all checkpoints, we simply update the device activation timestamp. Updating the account activation time allows the token to expire after seven days. This is one additional security layer that ScholarSeek does in order to protect our client's data.

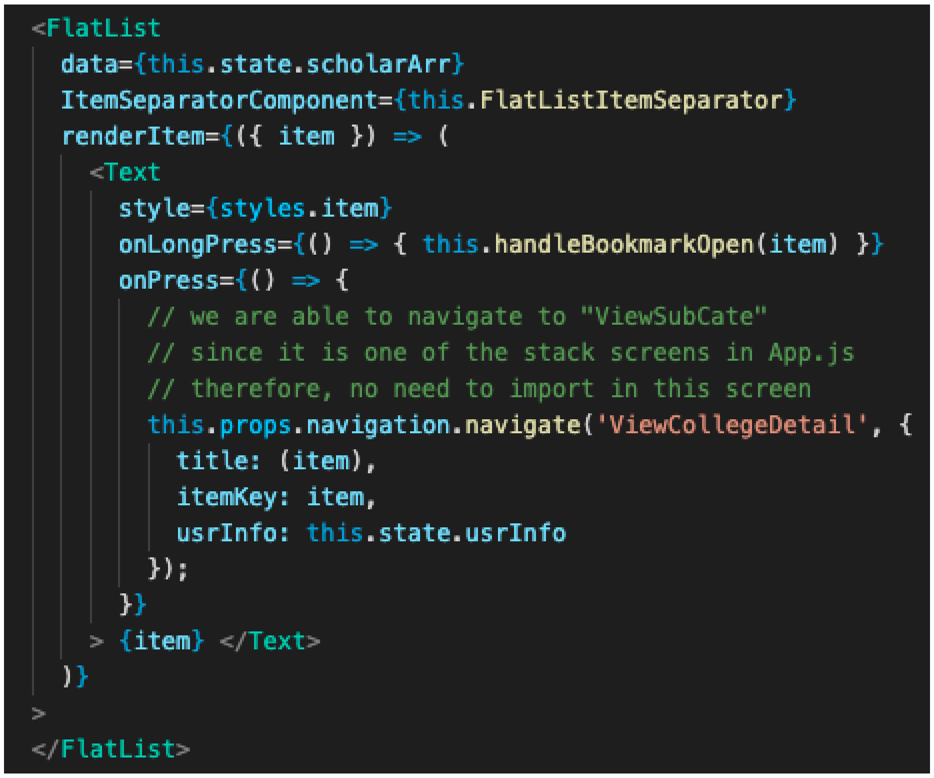
Categorized Browsing

ScholarSeek enables users to navigate Scholarship, college, and majors detailed information by categorized sections that allow the user easy to browse the most relevant scholarship, major, and college information as shown in the picture below.



(Figure 4.4-1, a screenshot of scholarship category section)

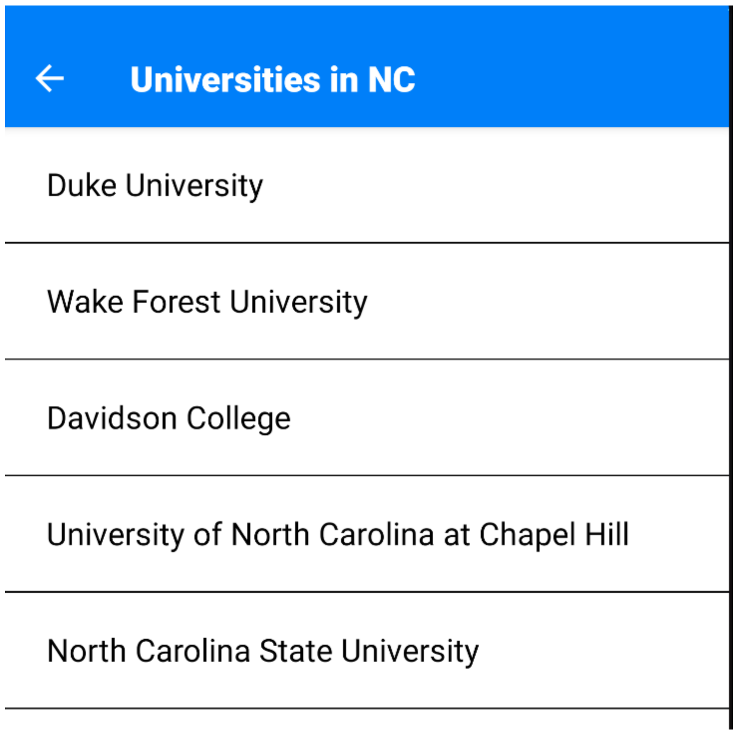
The category feature will vary throughout the scholarship, college, and major screen. For example, major categories have been categorized by state, such as New York, New Jersey, North Carolina, North Dakota, and Ohio. Then, users will be navigated to a sub-category which lists all the universities within a certain state, and if users want to browse to their desired university, they can click the university name and display full description of universities, which contain name of university, address, deadline, description, more details, ranking detail, and admission details. This implementation of logic has been applied throughout scholarship, college, and major.



```
<FlatList
  data={this.state.scholarArr}
  ItemSeparatorComponent={this.FlatListItemSeparator}
  renderItem={({ item }) => (
    <Text
      style={styles.item}
      onLongPress={() => { this.handleBookmarkOpen(item) }}
      onPress={() => {
        // we are able to navigate to "ViewSubCate"
        // since it is one of the stack screens in App.js
        // therefore, no need to import in this screen
        this.props.navigation.navigate('ViewCollegeDetail', {
          title: (item),
          itemKey: item,
          usrInfo: this.state.usrInfo
        });
      }}
    > {item} </Text>
  )}
>
</FlatList>
```

(Figure 4.4-2, a screenshot of code snippet FlatList dependencies)

We could have implemented the category feature with one of react native dependencies, it is called “FlatList”, allowing us to render multiple columns of the list dynamically. A matter of FlatList implementation, an array which is called “scholarArr” is instantiated in local state, and all applicable lists of information are retrieved by GET API call, which is formatted in an array of JavaScript Object Notation (JSON). Then, the array of JSON is assigned to the array of “scholarArr”. The scholarArry will pass to FlatList components in one of attributes, which is called “data”. Afterwards, the “renderItem” attribute will render a list of text components with item property as shown in the picture below.



Duke University

Wake Forest University

Davidson College

University of North Carolina at Chapel Hill

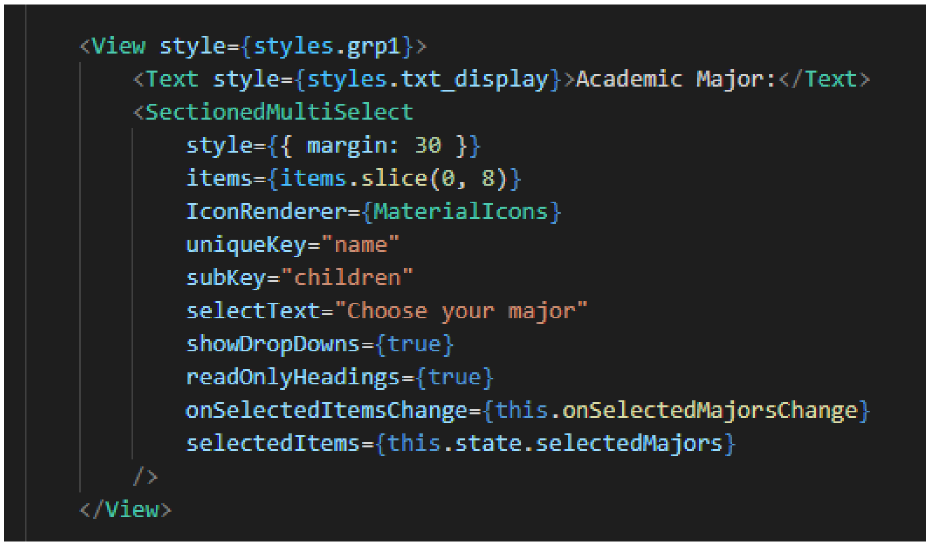
North Carolina State University

(Figure 4.4-3, a screenshot of implementation FlatLis dependencies)

As shown in the picture above, the “scholarArr” items are rendered in chronological order. Furthermore, the lines between each item are rendered by specifying react native styleSheet component property through the “ItemSeparatorComponent” attribute in FlatList corresponding to each the “scholArr” items.

User Surveys

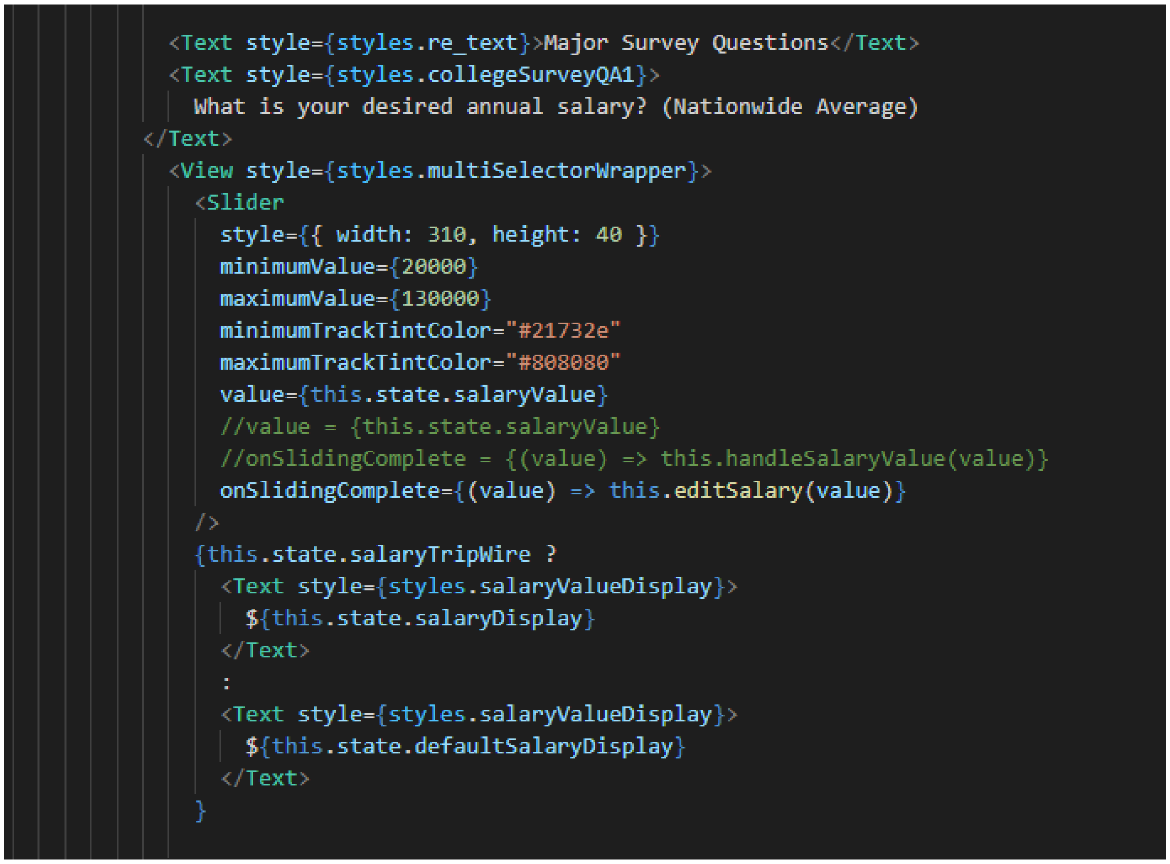
A key component in implementing the user surveys was the Sectioned Multi-Select dependency. This component enabled the user to make a selection from a searchable drop down menu compatible on both Android and IOS. The component consisted of a rendered element, pictured in the figure below, a method to append the local array containing the selected elements, and a constant containing an array within which all selectable terms were defined. This constant array of terms contained all multi-select fields in the scholarship survey and as such had to be indexed in the rendered component for the correct fields. This indexing can be seen in the items = {items.slice(0,8)} statement in the rendered component.



```
<View style={styles.grp1}>
    <Text style={styles.txt_display}>Academic Major:</Text>
    <SectionedMultiSelect
        style={{ margin: 30 }}
        items={items.slice(0, 8)}
        IconRenderer={MaterialIcons}
        uniqueKey="name"
        subKey="children"
        selectText="Choose your major"
        showDropDowns={true}
        readOnlyHeadings={true}
        onSelectedItemsChange={this.onSelectedMajorsChange}
        selectedItems={this.state.selectedMajors}
    />
</View>
```

(Figure 4.5-1, code snippet of sectioned multi-select component)

One of the more complex user survey front end implementations was the Major Survey. A challenge that was encountered in the development of the surveys was the lack of a suitable drop down menus that could be used for selecting a singular item from relatively few fields. While we had an almost suitable component in what was used for selecting gender in the scholarship survey, this component could not be used multiple times on the same screen without glitched behavior. Drop down components that were cross platform compatible were virtually non-existent. The sectioned multiselect was not suitable for selecting a single item from a few fields. In light of these circumstances, a slider was chosen as the most suitable component to allow the user to select a single option from three available options. The sliders returned numeric values which had to be parsed to be converted to the string values that would be stored on the backend and such that they could be displayed to the user as the current value for a given position on the slider. Below we can observe the slider element for expected average salary selection.

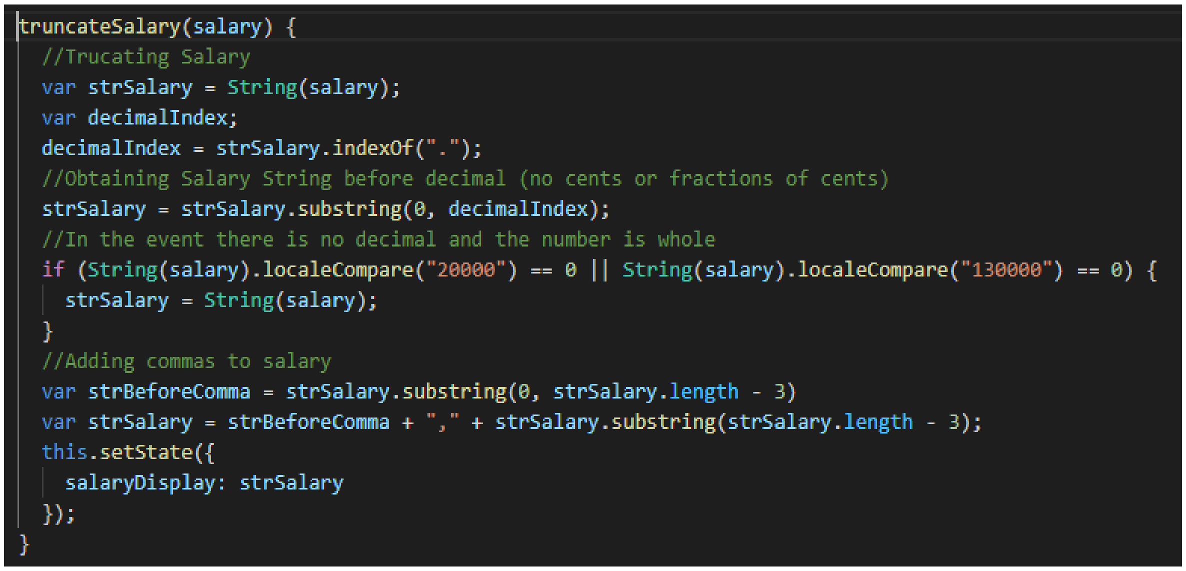


```
    <Text style={styles.re_text}>Major Survey Questions</Text>
    <Text style={styles.collegeSurveyQA1}>
      What is your desired annual salary? (Nationwide Average)
  </Text>
    <View style={styles.multiSelectorWrapper}>
      <Slider
        style={{ width: 310, height: 40 }}
        minimumValue={20000}
        maximumValue={130000}
        minimumTrackTintColor="#21732e"
        maximumTrackTintColor="#808080"
        value={this.state.salaryValue}
        //value = {this.state.salaryValue}
        //onSlidingComplete = {(value) => this.handleSalaryValue(value)}
        onSlidingComplete={(value) => this.editSalary(value)}
      />
      {this.state.salaryTripWire ?
        <Text style={styles.salaryValueDisplay}>
          ${this.state.salaryDisplay}
        </Text>
        :
        <Text style={styles.salaryValueDisplay}>
          ${this.state.defaultSalaryDisplay}
        </Text>
      }
```

.

(Figure 4.5-2, code snippet of major survey slider component)

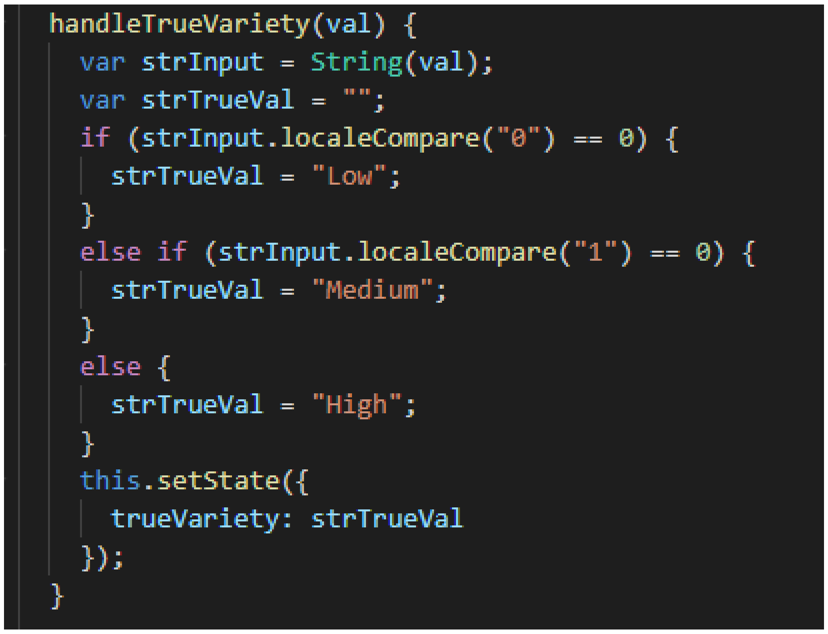
As the value of the slider is a float value, if it is displayed without any processing, it will be difficult for the user to read (ie 20000.000, if the user moves the slider ie. 20000.56775, etc.). Thus the value needs to be processed before it can be displayed. The following method is used to accomplish that.



```
truncateSalary(salary) {
  //Trucating Salary
  var strSalary = String(salary);
  var decimalIndex;
  decimalIndex = strSalary.indexOf(".");
  //Obtaining Salary String before decimal (no cents or fractions of cents)
  strSalary = strSalary.substring(0, decimalIndex);
  //In the event there is no decimal and the number is whole
  if (String(salary).localeCompare("20000") == 0 || String(salary).localeCompare("130000") == 0) {
    strSalary = String(salary);
  }
  //Adding commas to salary
  var strBeforeComma = strSalary.substring(0, strSalary.length - 3)
  var strSalary = strBeforeComma + "," + strSalary.substring(strSalary.length - 3);
  this.setState({
    salaryDisplay: strSalary
  });
}
```

(Figure 4.5-3, code snippet of component return value processing)

For the sliders that were used to answer survey questions whose answers were not numeric, the numeric output of the slider had to be converted to the correct backend value. An example of this function can be seen below. In this instance the slider returned a value of 0, 1, or 2 when asking the user for their preferred level of job variety in their field.

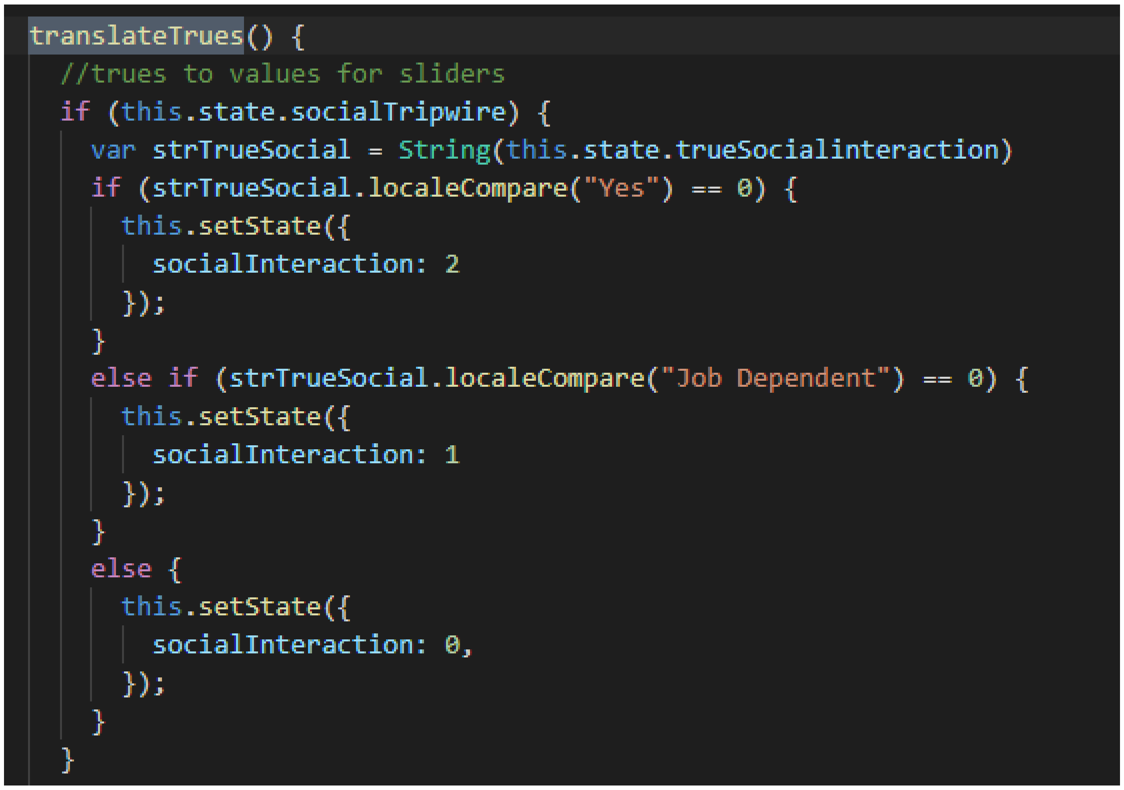


```
handleTrueVariety(val) {
  var strInput = String(val);
  var strTrueVal = "";
  if (strInput.localeCompare("0") == 0) {
    strTrueVal = "Low";
  }
  else if (strInput.localeCompare("1") == 0) {
    strTrueVal = "Medium";
  }
  else {
    strTrueVal = "High";
  }
  this.setState({
    trueVariety: strTrueVal
  });
}
```

(Figure 4.5-4, code snippet of converting returned slider output to values that can be sent to the backend)

These numeric values were then converted to their actual backend values and stored in the local state variable (delivered via API).

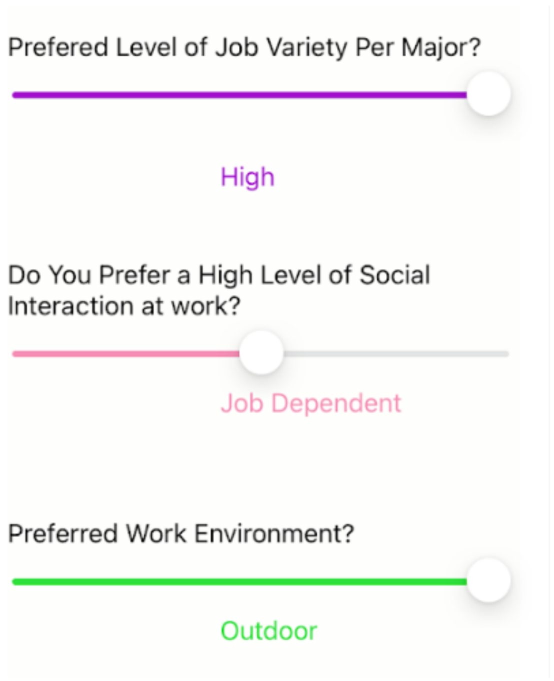
When receiving values from the backend, the values had to be converted to their numeric values to display the slider progress correctly. The following method was used to do this, executed at the end of the API call used to retrieve a user's existing survey.



```
translateTrues() {
  //trues to values for sliders
  if (this.state.socialTripwire) {
    var strTrueSocial = String(this.state.trueSocialinteraction)
    if (strTrueSocial.localeCompare("Yes") == 0) {
      this.setState({
        socialInteraction: 2
      });
    }
    else if (strTrueSocial.localeCompare("Job Dependent") == 0) {
      this.setState({
        socialInteraction: 1
      });
    }
    else {
      this.setState({
        socialInteraction: 0,
      });
    }
  }
```

(Figure 4.5-5, Code snippet of translating backend values to numeric values required to display correct slider state)

This method called at the end of an API call to fetch an existing survey for which these fields are present, converts the backend values into the correct numeric values which could be used by the sliders to display correctly.


Prefered Level of Job Variety Per Major?

High

Do You Prefer a High Level of Social Interaction at work?

Job Dependent

Preferred Work Environment?

Outdoor

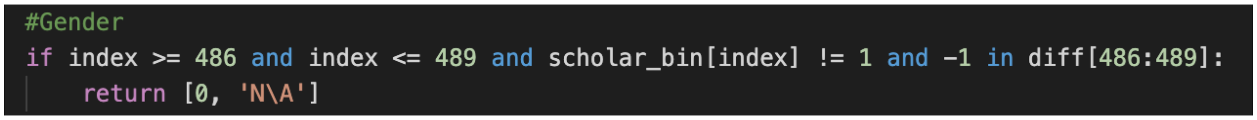
(Figure 4.5-6, screenshot of some major sliders)

<u>Personalized Recommendation</u>

Utilizing the client profile in figure x from the section Database Design, recommendations would be made off of the survey subdocuments. Each survey subdocument was utilized for their respective category recommendations model, having it be utilized to either query documents, or be used for calculations to evaluate the similarity of the document and the user. There are three recommendations models, for the 3 modules of this project: scholarship, college, and major

Scholarship Recommendation

First for the scholarship recommendation model, there will be a query utilizing the terms list to find scholarships that contain at least one input of the user's profile. Then, utilizing the user's numeric array, and a numeric array generated for each scholarship, there will be a comparison between the vectors. From the user's profile we will find all the indexes that contain a 1, since a 1 symbolizes that a user satisfies that specific value based on their input. This also allows a simplification of the vector dimensions to just focus on those that pertain to the specific user. After isolating the vector indexes that pertain to the user, we will extract those vector array indexes from each scholarship returned from the query, and go through a series of if statements as shown in figure x .



```
#Gender
if index >= 486 and index <= 489 and scholar_bin[index] != 1 and -1 in diff[486:489]:
    return [0, 'N\A']
```

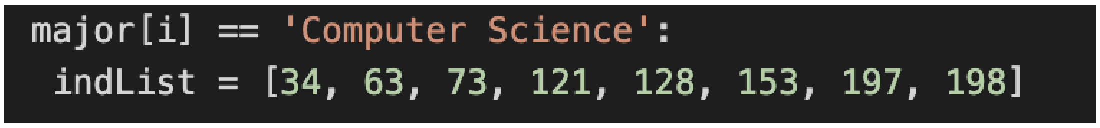
(Figure 4.6.1-1, Scholarship Recommendation Filtering)

The purpose of the if statement is to remove any scholarships that do not pertain to the user. How this if statement accomplishes this task is by the fact that a range of indexes represent all the answers for a specific survey question. As a result we break down the if statements into doing a check for each section of the array, if the scholar_bin (the scholarship vector array) at the index matching the user's profile is not a 1, this means that the scholarship doesn't contain this descriptor that the user has which is not bad. The third part of the if statement in figure _ however looks to see if the scholarship contains a value that the user does not for that survey section. Let's say if the user inputs male, and the scholarship does not contain a 1 for the male index in the vector array, it will check if any other gender specific array index contains a 1. If it does, that means that the scholarship does focus on gender being a deciding factor, thus making this scholarship invalid for the user to use since the user doesn't qualify for it based on their input. Every dimension of the scholarship vector will undergo this process to make sure that only valid scholarships are presented to the users.

After the filtering process is complete, the scholarship will be given a value, represented as: $\frac{\#\ of\ Scholarship\ Vector\ Dimensions\ \ Matching\ user\ Vector}{\#\ of\ UserDimensions\ Containing\ 1}$ as a decimal for ranking purposes, and as a fraction to send to the user to be able to see how many values of the user's profile match the scholarship. Utilizing the scholarships returned with a decimal value, the scholarships will be ranked in descending order, where the highest decimal value would be ranked first, since this implies that the scholarship is most similar to the user.

<u>**College Recommendation**</u>

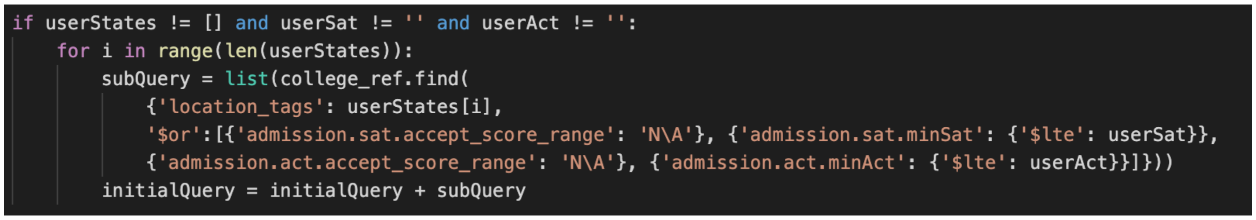
For the college survey, only the major attribute of the user's input turned into a numeric vector due to the fact that state and SAT/ACT score can be found through a simple query, while with major the database is not very cut and dry on what majors each college has. On top of that there are rankings of schools per major, and the popularity of each major in that college, so as a result we wanted to have the rankings have some sort of impact, so rankings will slightly increase the weight that a college specifically has for that major. Similar to the scholarship recommendation model, the numeric vector has each index representing a unique value. Whenever a user inputs a major, that major will be mapped to several indexes for the user's profile to represent that major in the numeric vector as shown in figure 4.6.2-1, setting those indexes to 1.



```
major[i] == 'Computer Science':
 indList = [34, 63, 73, 121, 128, 153, 197, 198]
```

(Figure 4.6.2-1, Indexing Specific Majors to Vector Array)

For the college recommendation, this is very dependent on if the user has inputted a major or not. This is due to the fact that only the major attribute is utilized for the numeric vector array, not the region or the ACT/SAT scores. The recommendation model checks what attributes the user has filled out, and compares it to a series of if statements, one example is shown in figure 4.5.2-2. If only the state and SAT/ACT scores are utilized, then the recommendation model will query return schools that satisfy both of those conditions.



```
if userStates != [] and userSat != '' and userAct != '':
    for i in range(len(userStates)):
        subQuery = list(college_ref.find(
            {'location_tags': userStates[i],
            '$or':[{'admission.sat.accept_score_range': 'N\A'}, {'admission.sat.minSat': {'$lte': userSat}},
            {'admission.act.accept_score_range': 'N\A'}, {'admission.act.minAct': {'$lte': userAct}}]}))
        initialQuery = initialQuery + subQuery
```

(Figure 4.6.2-2, Querying State, Sat & Major)

If the user's college survey contains a major input , then the recommendation model will also utilize a cosine similarity notated by the equation in figure 4.6.2-3 where the userCollegeVector is based off of the major input, and the collegeVector is based on the major information of that college from niche.com.

$$\cos(\theta) = \frac{userCollegeVector \cdot collegeVector}{|userCollegeVector||collegeVector|}$$

(Figure 4.6.2-3, Cosine Similarity)

The results of the cosine similarity will be utilized to rank colleges, based on their ranking in that specific major, as well as the popularity of that major based on how abundant are people graduating for that major in each specific college. After a cosine similarity value is attached to each college that resulted from the query of either region, test score, or both, the colleges will be ranked in descending order. This is to present the colleges that excel in the specific major(s) that the user has provided.

Major Recommendation

For major recommendation, the user's inputs will be taken, similar to the scholarship_survey, and transformed into a numeric vector array of 1's and 0's. How the major recommendation model works is that all the majors will be queried from the mongo db, this is because we want to compare all majors to the user's profile numeric vector array. Then we will utilize a cosine similarity again, similar to the college survey:

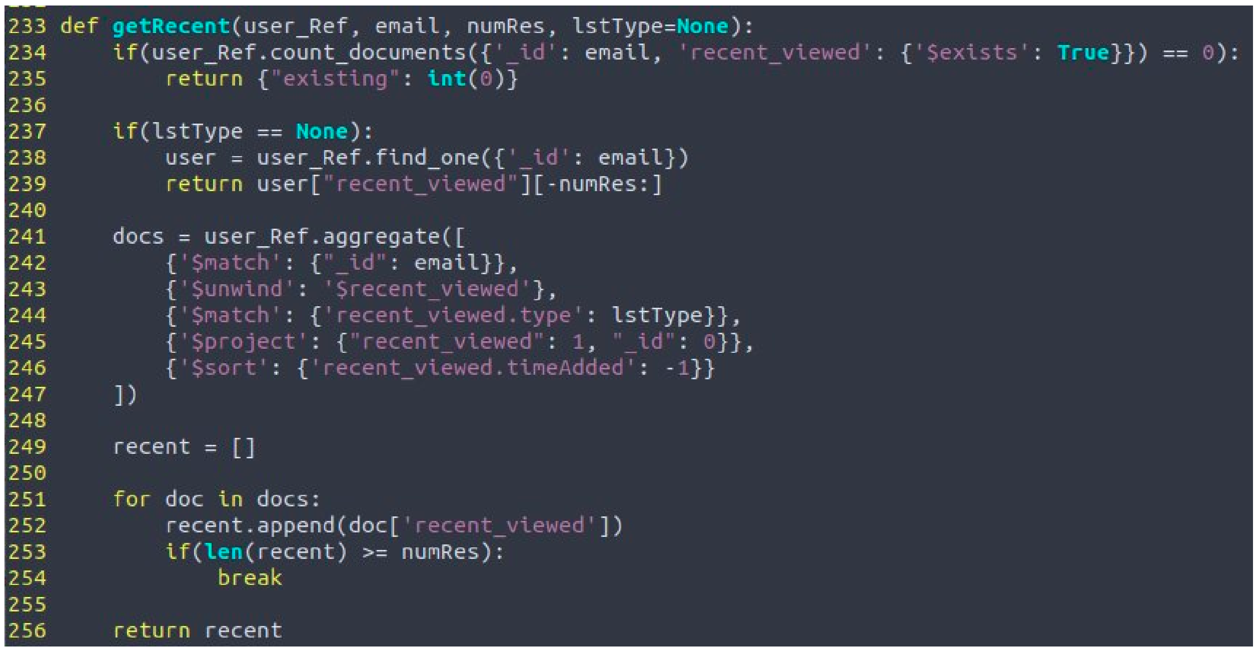
$$\cos(\theta) = \frac{userMajorVector \cdot majorVector}{|userMajorVector||majorVector|}$$

(Figure 4.6.3-1, Cosine Similarit)

With the results of the cosine similarity, we will group the majors together, based on niche.com's categorization of each major, and find the average of the cosine similarity value of each category, and rank the categories based on that value in descending order of the average cosine similarity value. Then the top category will be returned to the user. This is to tell the user the best fitting category of majors that would fit the user.

Instead of returning a ranked list of majors, we feel that a category would be best fitting due to the fact that majors within the same category shared common traits, and we did not want to pin the user to a single major. So for the purpose of this recommendation model, it's to tighten the scope of majors that a user may look into. This will save users time trying to look at such a brough range of majors to just a specific category.

Bookmarking & Recently Viewed

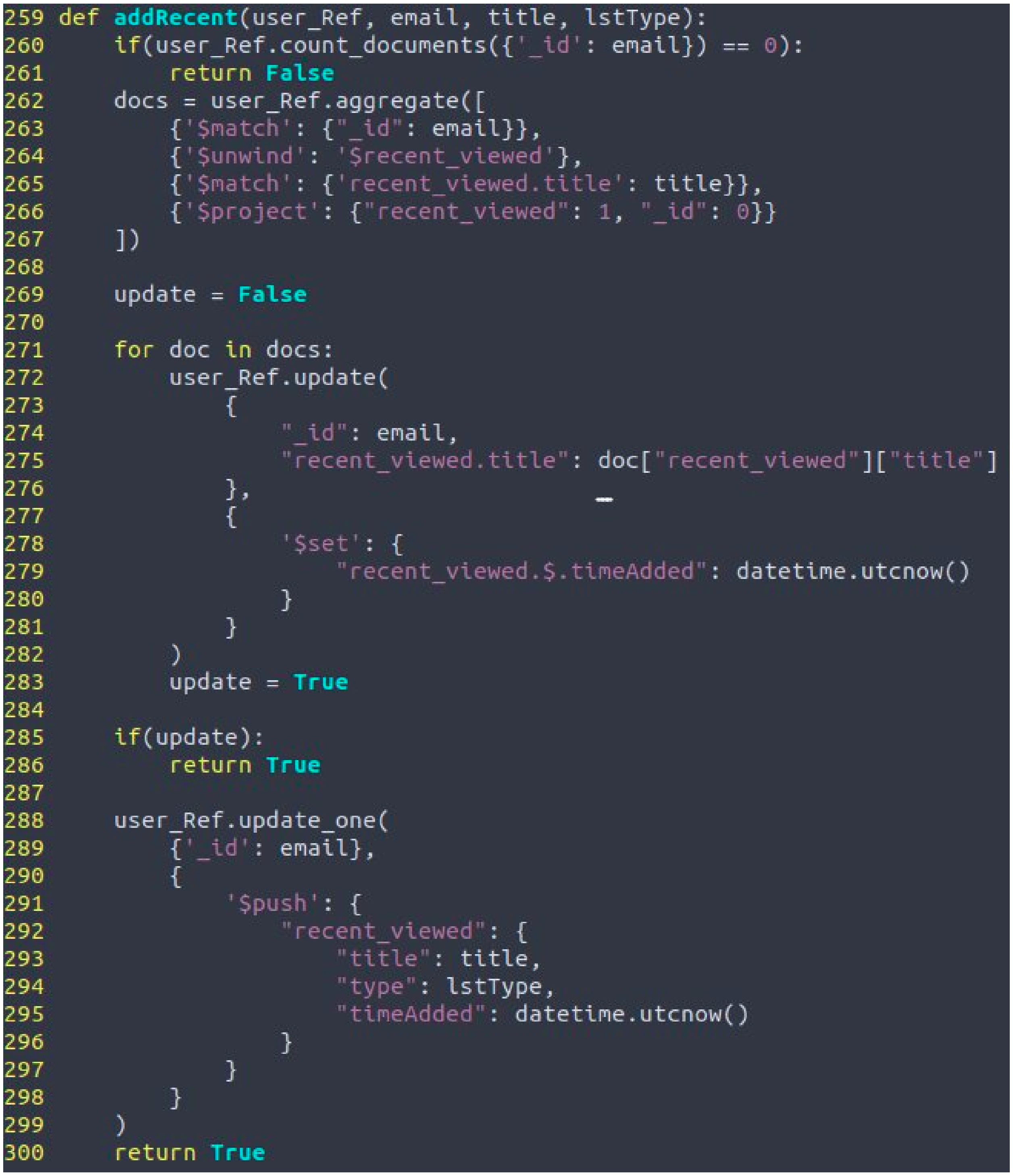


```
def getRecent(user_Ref, email, numRes, lstType=None):
    if(user_Ref.count_documents({'_id': email, 'recent_viewed': {'$exists': True}}) == 0):
        return {"existing": int(0)}

    if(lstType == None):
        user = user_Ref.find_one({'_id': email})
        return user["recent_viewed"][-numRes:]

    docs = user_Ref.aggregate([
        {'$match': {"_id": email}},
        {'$unwind': '$recent_viewed'},
        {'$match': {'recent_viewed.type': lstType}},
        {'$project': {"recent_viewed": 1, "_id": 0}},
        {'$sort': {'recent_viewed.timeAdded': -1}}
    ])

    recent = []

    for doc in docs:
        recent.append(doc['recent_viewed'])
        if(len(recent) >= numRes):
            break

    return recent
```

(Figure 4.7-1, Screenshot of the code to retrieve recently viewed)

This code gets the recent views for a given user. The numRes parameter signifies how many results the function should return, and lstType can either be set to None for all listing types, or specifically denoted as a major, college or scholarship to get on listings of a certain type. If the lstType is None then it just returns the last numRes results.

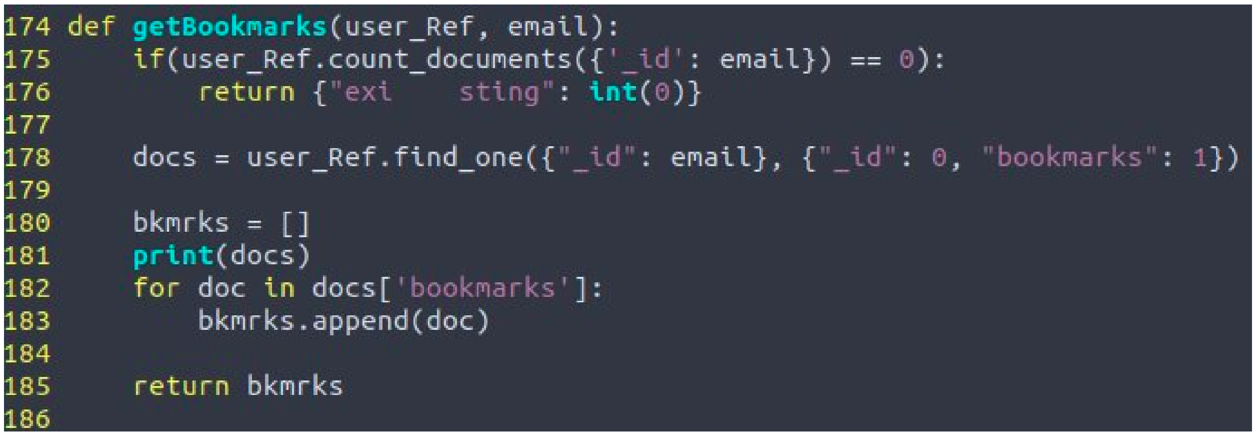
However if there is a specific lstType specified some filtering must be done, first and stored in a Python List before returning.



```python
def addRecent(user_Ref, email, title, lstType):
    if(user_Ref.count_documents({'_id': email}) == 0):
        return False
    docs = user_Ref.aggregate([
        {'$match': {"_id": email}},
        {'$unwind': '$recent_viewed'},
        {'$match': {'recent_viewed.title': title}},
        {'$project': {"recent_viewed": 1, "_id": 0}}
    ])

    update = False

    for doc in docs:
        user_Ref.update(
            {
                "_id": email,
                "recent_viewed.title": doc["recent_viewed"]["title"]
            },
            {
                '$set': {
                    "recent_viewed.$.timeAdded": datetime.utcnow()
                }
            }
        )
        update = True

    if(update):
        return True

    user_Ref.update_one(
        {'_id': email},
        {
            '$push': {
                "recent_viewed": {
                    "title": title,
                    "type": lstType,
                    "timeAdded": datetime.utcnow()
                }
            }
        }
    )
    return True
```

(Figure 4.7-2, screenshot of the code to store recently viewed pages)

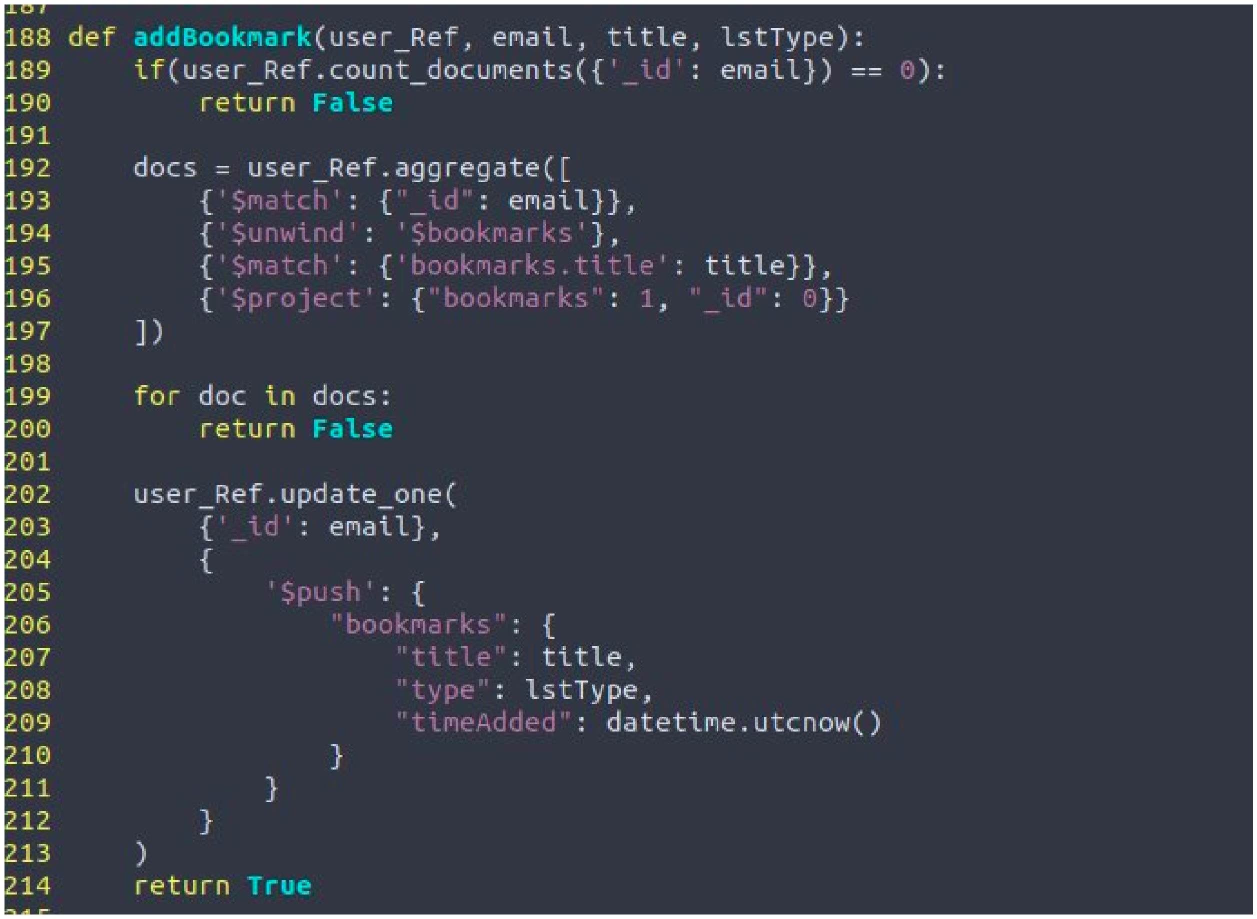
When a user visits a page, addRecent is automatically called, storing the user's recently viewed pages for later access. It takes in a title and list type, and checks to see if the document exists in the database already. If it does, it will update the time visited for all entries (there should only ever be one, since a listing with the same title is updated rather than added). If the entry does not exist in the database it is added.



```
def getBookmarks(user_Ref, email):
    if(user_Ref.count_documents({'_id': email}) == 0):
        return {"exi    sting": int(0)}

    docs = user_Ref.find_one({"_id": email}, {"_id": 0, "bookmarks": 1})

    bkmrks = []
    print(docs)
    for doc in docs['bookmarks']:
        bkmrks.append(doc)

    return bkmrks

```

(Figure 4.7-3, screenshot of the code to retrieve bookmarks)

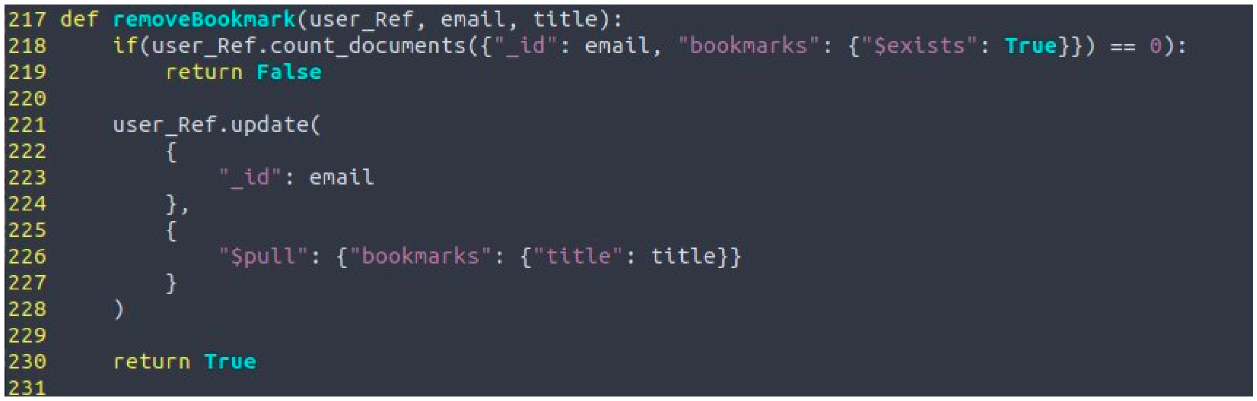
If a user chooses to get bookmarks, a Python List of all bookmarks for that user is collected and returned.



```python
def addBookmark(user_Ref, email, title, lstType):
    if(user_Ref.count_documents({'_id': email}) == 0):
        return False

    docs = user_Ref.aggregate([
        {'$match': {"_id": email}},
        {'$unwind': '$bookmarks'},
        {'$match': {'bookmarks.title': title}},
        {'$project': {"bookmarks": 1, "_id": 0}}
    ])

    for doc in docs:
        return False

    user_Ref.update_one(
        {'_id': email},
        {
            '$push': {
                "bookmarks": {
                    "title": title,
                    "type": lstType,
                    "timeAdded": datetime.utcnow()
                }
            }
        }
    )
    return True
```

(Figure 4.7-4, screenshot of the code to store bookmarks)

If a bookmark is added the list of bookmarks is collected, and if the bookmark is in the database the function is exited before it is placed in the database. Unlike with the history there is no need to update the timestamp because we want to maintain the time that the user initially bookmarked the page. This should never be an issue because on the front end a user won’t be able to double bookmark a page, but it’s included here in case an error occurs. On the other hand, if the entry does not exist it is added to the database.

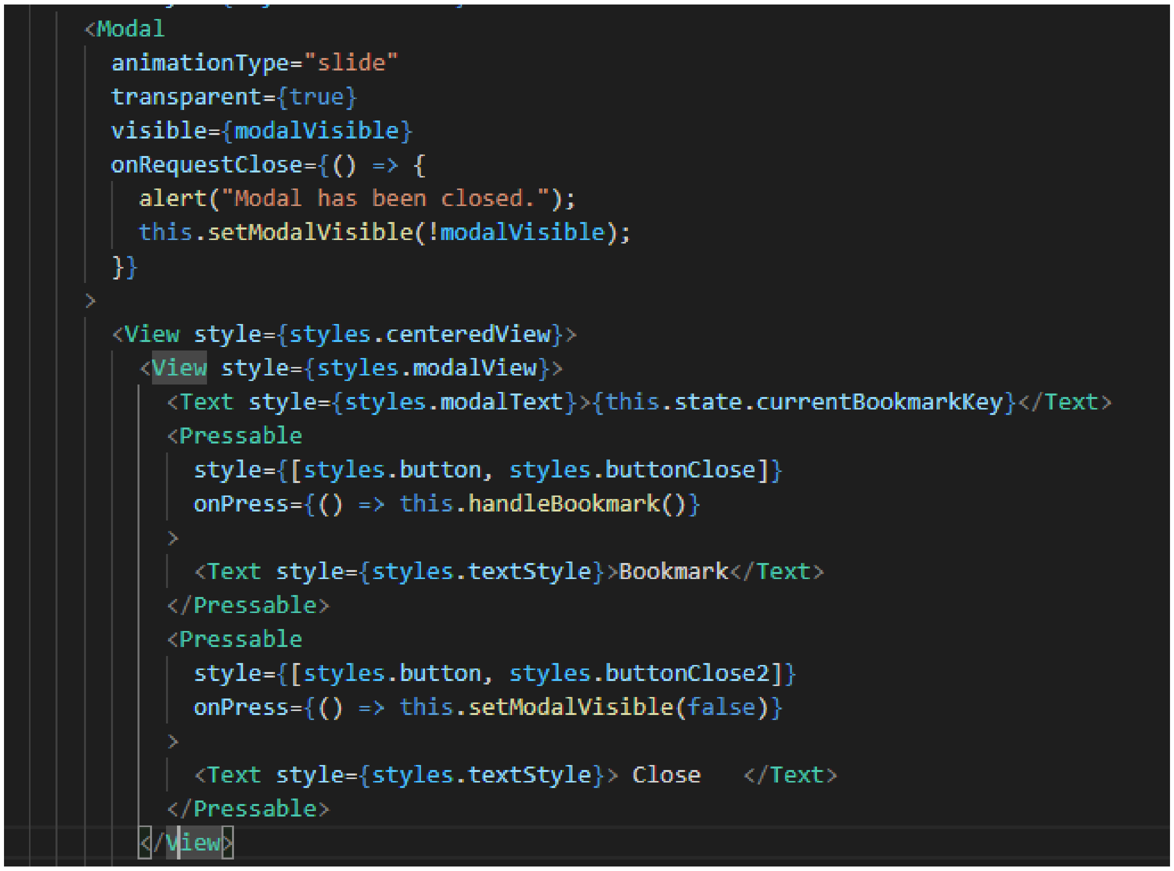


```python
def removeBookmark(user_Ref, email, title):
    if(user_Ref.count_documents({"_id": email, "bookmarks": {"$exists": True}}) == 0):
        return False

    user_Ref.update(
        {
            "_id": email
        },
        {
            "$pull": {"bookmarks": {"title": title}}
        }
    )

    return True

```

(Figure 4.7-5, screenshot of the code to remove a bookmark)

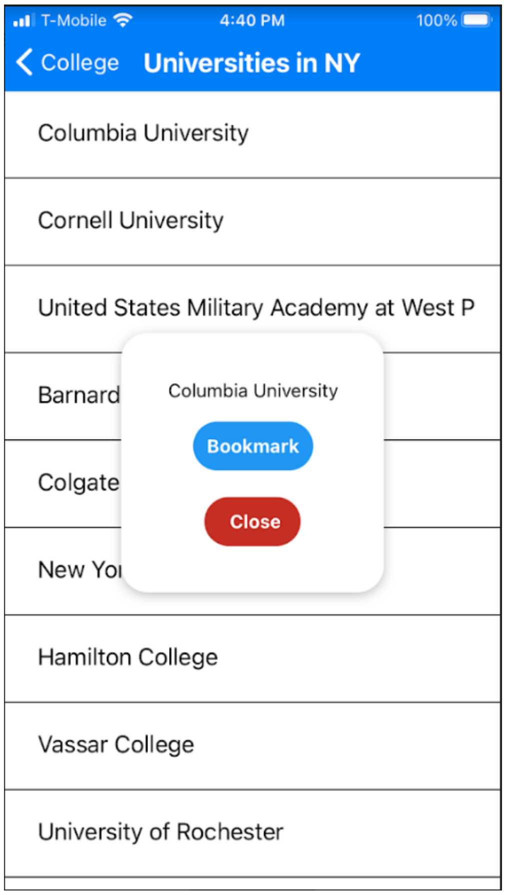
If a user would like to remove a bookmark this function is called. It updates the database removing bookmarks where the title of the document matches the title passed in.

For the frontend for the bookmarking feature, two methods were implemented to allow for bookmarking. The first was a long press initiated pop up modal from feature category listing pages. This feature can be accessed from anywhere in the app where the user is presented a list from which to navigate to a specific listing. The modal and pressable react components were used to implement this feature. The modal relied upon a local variable known as modalVisible to determine whether or not it should be shown to the user. A method was required to assign the local variable boolean values in order to toggle the modal. In the touchable opacity component within the flatlist component (used to render the listings), upon a long press by the user, a method is called to set the modal visibility to true and to save the selected title of the bookmark from the iterable listing. The modal then opens, within which there exists a pressable component which handles either the submission of the listing information to the bookmark API call or simply closes the modal.



```
<Modal
  animationType="slide"
  transparent={true}
  visible={modalVisible}
  onRequestClose={() => {
    alert("Modal has been closed.");
    this.setModalVisible(!modalVisible);
  }}
>
  <View style={styles.centeredView}>
    <View style={styles.modalView}>
      <Text style={styles.modalText}>{this.state.currentBookmarkKey}</Text>
      <Pressable
        style={[styles.button, styles.buttonClose]}
        onPress={() => this.handleBookmark()}
      >
        <Text style={styles.textStyle}>Bookmark</Text>
      </Pressable>
      <Pressable
        style={[styles.button, styles.buttonClose2]}
        onPress={() => this.setModalVisible(false)}
      >
        <Text style={styles.textStyle}> Close   </Text>
      </Pressable>
    </View>
```

(Figure 4.7-7, code snippet of pop up modal component)



(Figure 4.7-8, pop up modal UI/UX)

<u>Security</u>

As the main feature of the app revolves around collecting possibly sensitive user data from the user, one of the issues which was very important to the team was the protection of that data. One aspect of this was implemented in the user authentication system. Upon signing up a user's password is immediately salted by adding a secret key in front of the password, in order to protect against brute force attacks. The password is then hashed using the MD5 algorithm before being stored in the database. When a user attempts to sign in to the app, the same process is repeated on the entered password, and compared to the value in the database. In this manner there is no decryption, and the real password is never stored anywhere on the system. Adding an extra layer of protection, a user must confirm the email they entered on sign up is legitimate by clicking on a link sent in an email from our system, ensuring the user is who they say they are.

Another consideration was protection from potentially malicious actors who would attempt to intercept the data as it was being transferred from the frontend to the backend and vice-versa. In order to combat this, JWT was used to encrypt the user's device token using the HS256 algorithm, and decrypting the data using a 256-bit secret key. The 256-bit secret key is stored in our database where it never sends to the frontend. The JWT also has a universally unique identifier (UUID) associated with it, and a seven days expiration date to ensure that no malicious data is sent from an unauthorized source. The expiration date is a timestamp in the UTC time zone, which will be stored on a database along with the UUID. When a request is made, the backend compares the JWT code that is associated with the user's profile. Then we utilize the 256-bit secret key to decrypt the JWT in order to compare the expiration timestamp. If any process failed during the decryption and comparing, we will not authenticate the user to access any resources.

Testing

The testing we did for the scholarship recommendation system was seeing how our recommendation model worked under different conditions, in this case comparing a different amount of survey values imputed, ranging from as little as 0 where the user simply browsed by a single category, to recommending a user with 21 attributes. As shown in figure x, a user who simply browsed the scholarship tab looking through scholarships pertaining to just their state was only able to find a single scholarship that is applicable to them. However if they try to do a recommendation without state, in the top 10 documents none of them are relevant. This shows how important state is as an attribute to the recommendation model. Then as we progress with adding attributes, the precision increases as well as the precision mean average, meaning that the position of relevant documents on average are located higher. Based on this test, it concludes that the recommendation does have an impact on the precision of documents returned, as well as having a rather direct correlation between the number of user dimensions (inputs) and the precision as well as the mean average precision.

	Browse By State	3 User Dimensions (No state)	4 User Dimensions	6 User Dimensions	10 User Dimensions	21 User Dimensions
Precision	0.1	0	0.5	0.5	0.7	0.9
Mean Avg Precision	1	0	0.419	0.745	0.776	0.989
Records Returned (Max 10)	10	10	10	10	10	10

(Figure 5.0-1, Scholarship Recommendation Results)

For college recommendation we wanted to test and see how the combination of attributes between college, test scores(SAT/ACT), and major impacted the precision and mean precision average of the records returned from the recommendation algorithm. What was categorized as a relevant college was a college that would most likely be applicable to the user given what they have already supplied. When referring to figure x below, you can see that when not using any college recommendation, there are no documents applicable to a user. This is due to the fact that niche orders the colleges based on their ranking, making the default ordering within the state category, the highest ranking and thus also the highest admission bar . As a result, users would be more inclined to utilize the algorithm to cut through the prestigious colleges and be recommended/given colleges. One may also note within the table that the only situation where the precision is not 1 is where the survey only contains the state and major. The reason for this is due to the ACT/SAT score being the deciding factor for if the college is applicable or not to the user. The reason for still including this combination to be allowed is due to if a user hasn't taken the SAT or ACT yet but would like to have their college recommended, this feature would still be available to them

	Browse By State	State/Sat & Act	State/Major	Sat & Act /Major	State/Sat & Act/ Major
Precision	0	1	0.6	1	1
Mean Avg Precision	0	1	0.68	1	1
Records Returned (Max 10)	10	10	10	10	3

(Figure 5.0-2, College Major Recommendation)

Lastly for the major recommendation testing, the only metric to truly test this recommendation algorithm is based on if a user is satisfied with the results of their survey. As a result the five of us took the major recommendation survey and as a result 3 out of the 5 were satisfied with the results of the recommendation. With over a 50% success rate, and such a minimal sample size, we have decided to not modify the major recommendation. After more testing with a larger sample size, if we see that people are more often than not satisfied with their recommendation, the next steps would be modifying the weights of certain survey questions to show that they have more weight than others.

<u>Conclusion</u>

<u>Limitations</u>

One of the biggest limitations that the team faced was the difficulty of finding a dataset which fit the needs of the project. Every dataset that the team examined seemed to have some flaw with it. Many of the sets were missing crucial information, had some biases associated with them, or were out of date. This caused problems in the development process because the data was the most crucial part to the efficacy of the app.

Ultimately the data was sourced from several different datasets, and aggregated together, which increased the complexity and cohesiveness of the data.

The order in which React Native updates local variables and renders components and our limited understanding of this process led to some interesting issues. Dynamic error checking (live feedback on correctness of input as the user types it for the surveys) , while mostly functional, had to be deprecated due to a bug where once a correct GPA or exam score had been given, all but the last digit could be deleted. We are confident this was a life cycle issue as it could be observed that as soon as the last digit was deleted it reappeared. Ultimately the error checking logic was moved to form submission to avoid this issue. Provided more time, this issue could likely be resolved. Another example of the React Native life cycle providing a challenging limitation was the inability to re-render screens upon navigation to another page dependent on input on that page. For example, if the user navigates from the bookmarks page to a specific bookmark they have listed, removes the bookmark and then navigates back to the bookmarks page, the bookmark they have removed is still present in the listing until the user navigates to the bookmark page from the account screen a second time. This is because when navigating back to the bookmarks page from a listing, React Native uses the component that has already been rendered. We attempted to use a React Native functionality known as UseIsFocused to force a re render of the screen if the user navigated away from the screen. This solution did work however it was far from ideal as it continually executed rendering until the user navigated back to the bookmarks screen. Because of this infinite loop scenario the solution was abandoned. The other solution to this issue, the use of navigation events including listeners (and hence navigation props) did not work either and this issue bleeds into another limitation, which was the passing down of props in React Native.

Another limitation that we have confronted during frontend development was passing down property between react navigation components. In our project, our react native components were constructed in multiplex hierarchical order. First, we acquired a user's email, JWT token, and unique user's device ID from the parent component, App.js, and the user's property which includes user's email, JWT token, and unique user's get passed down to another child component to call applicable GET API. However, some components resided inside of sub-child components which also inhabit inside of another sub-child component. We had to pass down the property one by one in react native component hierarchical order, which made it hard for us to keep track of where user property was passed down from. Therefore, a better alternative solution that we came across was one of react native libraries which was called "Redux" or "Context API", but since we are permitted within development time limits, we rather pass down the user property between parent and child components.

Another limitation was a lack of external users that could test the app. It was difficult for the team to gauge the effectiveness of the recommendation models without having a group of users to test the app on and gather feedback from.

<u>Future Enhancements/Recommendation</u>

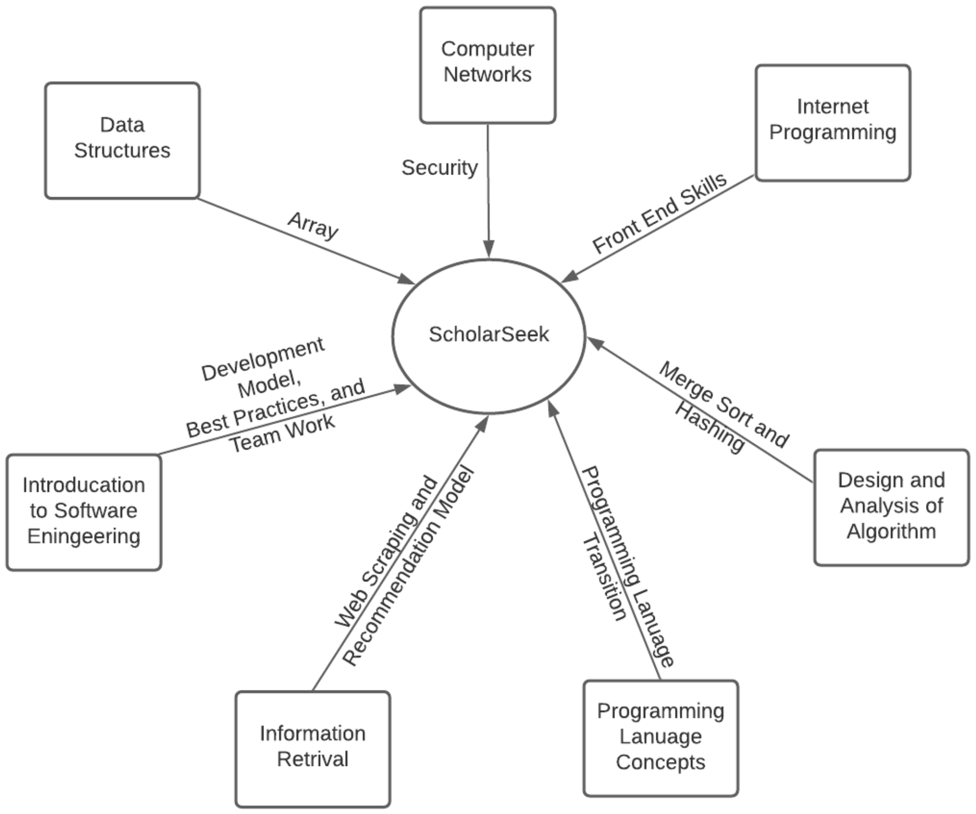
There are currently several items on the list for this project if development was to continue in the future. We would add one of the DevOps practises into our project, Continuous Integration (CI), which allows our team to have testing before pushing the commit(s) into the code repository’s main branch. For instance, utilize a CI tool to have unit or functional testing our code before the git push. If any testing fails, we would instantly get notified. With CI, it will speed up our Software Development Life Cycle with minimum error as possible. Furthermore, we would like to add encryption on the frontend for the user’s device token. A possible solution for this is to use a secure storage API for React Native as we were suggested by the author of React Native Expo (https://github.com/expo/expo/issues/12083). The secure storage API allows our app to store the device token securely stored and encrypted on the user’s phone so that other apps will not be able to access it. Another security consideration that was looked into was encryption of the entire database. This would make it much more difficult for would-be hackers to collect users information. By encrypting the entire database, getting just one piece of information about a user would become difficult, and getting enough information to do anything malicious with it would be nearly impossible. This would secure our system from damaging data leaks, and ensure the maximum protection of user data.

There are also several features that would improve user experience which are slated for development in the future. One feature would be an account deletion option. Once a user has found their scholarship, college and/or major, they may no longer have need of ScholarSeek and don’t want their data to be stored any longer. An account deletion option would be a good solution to this problem, making sure a user's data is not stored extraneously. We also wanted to ensure that users are aware of how to use our app to its fullest extent, and so another feature that the team would like to implement is including Tooltips on a user's first sign in explaining how to use the app and walking them through a small tutorial.

Accessibility is a huge concern for modern applications and the team believes that ScholarSeek should follow these guidelines. In the future ensuring that accessibility features such as text-to-speech function seamlessly within the app is a high priority. Another important future feature would be implementing a feedback system to our recommendation model. This feature would allow users to report how well a recommended document fit their possible needs, which would allow the system to learn from it's successes and failures and make better recommendations in the future. Similarly in the future the team would like to incorporate other datasets which would fully populate missing data from some of the information that is stored on the database, mostly missing SAT/ACT requirements for some colleges. By incorporating a supplementary dataset, the system would have more data to use, and a better recommendation can be made. On top of new sources of information, we would like to make our database dynamic, to keep up to date information in regards to colleges, majors, and scholarships as well as removing any outdated information. The app is truly only useful if it contains the most accurate information for our user to utilize.

One final consideration for future features would be an administrator portal, where a member of the ScholarSeek team would be able to go into a portal and explore user data, as well as see bug reports, and user grievances. This portal would allow reports to be generated and analysis of data to see information like user retention, and what parts of the app a user is most interested in. In the future this would allow the team to have a high level of information regarding our app, and what features are most important to focus attention on.

Previous Course Contributions


Computer Networks
Security
Data Structures
Array
Internet Programming
Front End Skills
ScholarSeek
Development Model, Best Practices, and Team Work
Introducation to Software Eningeering
Merge Sort and Hashing
Design and Analysis of Algorithm
Web Scraping and Recommendation Model
Information Retrival
Programming Lanuage Transition
Programming Lanuage Concepts

(Figure 6.3-1, a screenshot of course previous course contributions)

The previous courses are helpful for us in order to make this project successful. For instance, we learned how to web scraping and build a recommendation model via Information Retrieval. Then we learned programming language transition from Programming Language Concepts, which was helpful for us to transit from Python and JavaScript. There are some algorithms involved such as merge sort and hashing which we learned from the Design and Analysis of Algorithms course. Since this is a mobile app, a UI is essential, the Internet Programming helped us a lot in terms of UI/ UX design. Also, the Computer Networks course helped us to get a better understanding of security and what we should include when designing our authentication system. The Data Structure course helped us to understand array data type, which allows us to understand MongoDB easier. Lastly, the Development Model, Best Practices, and Team Work that we learned from Introduction to Software Engineering are essential because it helps the team to communicate better and keep track of the progress of the project much easier.

Team Members

Michael Trzaskoma

- Contribution: Recommendation Algorithm Design, Database digitalizing to Numeric Vector
- What I learned: How to utilize mongodb, querying from the database, inserting documents, as well as updating existing documents. Learning JSON format of nested dictionaries and the limitations of pymongo with updating nested dictionaries resulting in having to update utilizing all the fields in the update function. Handling front end inputs and communicating to understand what will be sent from the front end to back end in order to know how to handle such inputs. Lastly is team management and communication as a whole was key to have modules of the project become completed in a timely manner where everyone was on the same page.

Hui (Henry) Chen

- Contribution: API Design, Web Scraping, Token, UX/UI design, Deployment
- What I learned: How to handle a reCAPTCHA system in a website when scraping data, especially with niche.com where we scraped a large-scale dataset. Handling a reCAPTCHA system was something that I did not expect as I assumed that my program was undetectable from it as I have a random timer pause on it. Also, I learned how to structure the Flask application so that it is deployable to Amazon Web Service EC2 through Docker. Lastly, I learned the importance of communication between each

member of the team because once there was a miscommunication between the frontend and backend on what kind of input or output will be for the APIs. Therefore, I realized the importance of communication as a key to push the project forward.

<u>Gregory Salvesen</u>

- Team Contributions: API Design, Security, Authentication, Team Website
- What I learned: From a technology standpoint I learned a lot about Flask and using Python as a web development tool. I became familiar with communication between a frontend system and a backend system using REST, and proper techniques for encryption. I didn't know anything about MongoDB and so had to learn it and how to use it from scratch. I became more familiar with Git and version control throughout the development process.
  As for non-technical skills I learned how to manage the workload on a large scale project and work within a team. I think that everyone on my team was very civil in their discussions and solutions to problems were handled swiftly and amicably. I believe this is a good framework for how discussions within a project should be and I learned how to communicate better because of it.

<u>Zakaria M. Khan</u>

- Team Contribution: Implemented UX/UI design, working especially closely with the surveys, bookmarking, and input handling and respective API calls
- What I learned: While having worked with React Native previously on a CSCI 380 (Introduction to Software Engineering) project, I had not delved as deep into how props are passed between screens in React Native including how to structure components to enable a successful front end data flow to accommodate navigation between screens. I also learned the differences between functional and class react components and how to

leverage both and convert components from one to the other depending on the circumstance. Previously I might have been stuck if a dependency was documented as a functional component but my own application was for a class component. Now I can switch between the two as needed. I also learned much more about the React Native lifecycle as with the editable surveys and other similar features, Scholar Seek required many live updates to the UI as input changed. I also became aware of how much more research, study, and practice I need to better understand the backend and how to integrate the backend outside of making API calls in the front end. Thanks to my group mates, I was introduced to how an API can be designed and how a recommendation model can be created for web scraped data. I also became more adept at reading and understanding documentation for new libraries and dependencies that I had not previously used and trouble shooting them when issues arose. My ability to research new dependencies to use as solutions for problems also developed from this project.

Jungi Park

- Team Contribution: assisted UX/UI design, Call API, Fixed UI bug.
- What I learned: The most valuable experience that I absorb from this project is what technologies to be used in order to develop fully functional applications because I have not had many experiences where I have been able to develop an application myself or as a team. However, due to talented team members, Henry, Gerg, Zak, and Michael, who have superior knowledge on application development, I learned how backend and frontend are communicated in order to be functional, which allows me to envision work as frontend developer in the near future. Also, I took a course where I learned how to design UI/UX design and call an applicable API to frontend by following its course material, but I did not know how to apply the knowledge that I learned from course to real world application. This project also gave me a direct insight into the effective

software development learning process as well as importance of communication between frontend and backend developers.

Bibliography

1. Existing Project from CSCI 426, https://github.com/hchen98/csci426-project
2. Scholly, https://myscholly.com
3. Scholarships.com, https://www.scholarships.com
4. CollegeBoard, https://bigfuture.collegeboard.org/scholarship-search#!personalinformation
5. Niche company profile: Valuation & investors, https://pitchbook.com/profiles/company/55277-02#overview
6. Niche, https://www.niche.com/about/
7. Information Retrieval - Final Report, https://docs.google.com/document/d/1_i6CTVkto8V1ejx1wWrfS-NTDv1kcDih2kcSCw19uZk/edit#heading=h.gjdgxs
8. Structure the backend with Flask, https://pythonise.com/series/learning-flask/your-first-flask-app
9. Mongodb with Python, https://pymongo.readthedocs.io/en/stable/
10. JWT, https://jwt.io/introduction/