Final Report

4 May 2022

Team Truthseeker

Garry Ancheta
Georgia Buchanan
Jaime Garcia Gomez
Kyler Carling

Project Sponsor	*Team Faculty Mentor*
NOBL Media – Jacob Bailly	Felicity H. Escarzaga

Table of Contents

1- INTRODUCTION 1

2 - PROCESS OVERVIEW 3

3 - REQUIREMENTS 6

4 - ARCHITECTURE AND IMPLEMENTATION 11

5 - TESTING 23

- UNIT TESTING 23
- INTEGRATION TESTING 28
- USABILITY TESTING 32

6 - PROJECT TIMELINE 35

7 - FUTURE WORK 37

8 - CONCLUSION 39

9 - GLOSSARY 41

APPENDIX A 42

1 - Introduction

Today, misinformation is widespread on the internet. Different platforms intentionally spread misinformation to harm individuals and groups. The internet, however, is just another representation of many businesses through websites. For most businesses, websites are another source of income through the showing of advertisements. Demand Site Platforms (DSPs) are what allow placement of advertisements on websites and currently, DSPs deal with misinformation by blacklisting or demonetizing websites, but only do so when they are actively told by the advertiser. Thus, an advertiser can be damaged by being associated with misinformation when their advertisement is placed on the wrong website; an example of this would be a police department recruitment advertisement that shows up on a far-right website. In this case, some people might take this as though the police department is specifically recruiting people with far-right ideology, when in reality, they are not. Not only does this harm the police department's reputation, but it also wastes money due to the fact that the advertisement is having the exact opposite effect of what the advertisement is intended to do.

Fortunately, NOBL Media has taken the initiative to prevent this. NOBL Media has developed a proprietary artificial intelligence which allows preemptive prevention of advertisements from appearing on certain web pages. NOBL Media's artificial intelligence scans web pages and rates how credible each web page is through linguistics. Using this service, advertisers can set a rating threshold through NOBL Media and then NOBL Media will take care of the rest: scanning web pages, and preventing the advertisers' advertisement from showing up on web pages that are below the set rating threshold.

However, NOBL Media does not have a way for their customers to visualize or obtain this data in any way and therefore, their customers cannot see the value in NOBL Media's service. This is where Team Truthseeker has aided NOBL Media through the creation of a web application and an Application Programming Interface (API). These two components allow NOBL Media to solve the problem with their business flow through the implementation of the following features:

1. The web application must be able to create and authenticate customer accounts to allow secure access to the customer's data using integrated technology Auth0.

2. The API must be able to handle user authentication requests. Once authenticated, the API must handle a request to be parsed into NOBL'S MySQL database.

3. The web application must be able to abstract JSON data coming from the NOBL Media MySQL database and represent these to customers through graphs and charts using technologies such as ECharts.

4. The API must be able to retrieve the JSON data from the NOBL Media MySQL database and return an HTTP 200 level response with JSON data to NOBL's web application or the client's own site.

5. The web application must allow customers to download customer ad data in a formatted file such as in a CSV or Excel file.

Once that information is collected, advertisers will be able to see how their advertisement is performing in terms of supporting misinformation and how much money NOBL Media is saving the advertiser.

2 - Process Overview

This section describes the different parts of Team Truthseeker's process of planning and implementing the project. This section will walk through different tools that were used to keep track of the team's progress as well.

Version Control

The team used Github for version control for an easy interface for using Git. Additionally, some members of the team specifically used Github Desktop as well to further simplify the team's version control. For most of the contributions towards the API and the Web Application, the team used pull requests to request a review for individual contributions. Furthermore, the team also created different branches for further development or whenever there needed to be a study of different implementations that the team had thought of.

Task Manager

The team used Trello to keep track of tasks. Every week, the team convened at the beginning of the week and made use of uploading the different tasks that were designated to be completed within the current week. Each team member was responsible for updating the status of the tasks assigned to them. The team lead, in the days before the end of the week, would check the Trello board to see the progress of each task and will communicate with each team member individually should a team member fall behind.

Roles

There were 7 roles in total for the team:

1. Team Lead
2. Customer Communicator
3. Editor
4. Architect
5. Coder
6. Release Manager
7. Quality Assurance Manager

Team Lead

- The Team Lead organizes each week and ensures that there are tasks that can be completed in the current week for each team member. The Team Lead also ensures that the meetings are held and ensures that the agenda for the meeting is followed. Additionally, the Team Lead also aids other team members should they fall behind in their tasks.

Customer Communicator

- The Customer Communicator is the team member who is responsible for relaying information to the team's client. Additionally, the Customer Communicator is also responsible for getting the different deliverables signed should the deliverable need the client's approval.

Editor

- The Editor is the team member responsible for revising and ensuring that deliverables have the required sections. The Editor also notifies individual team members if the team member's assigned section(s) on the deliverable needs improvement.

Architect

- The Architect is the team member responsible for creating the workflow of the project's different components and how the components communicate with one another. This role is responsible for planning the guidelines of how the project is to be built and ensures that it is followed during implementation.

Coder

- This role is universal within the team. However, this role is for those who are coding and actually implementing the project.

Release Manager

- This role is the team member responsible for ensuring that each push, pull request, and branch into the two components of the project are of good quality code and documentation.

Quality Assurance Manager

- The Quality Assurance Manager is the team member responsible for going through the different deliverables and code being pushed. Basically, an assistant and a second eye for the Release Manager and the Editor.

3 - Requirements

Functional Requirements

Functional requirements are what the project is expected to perform for the user. These requirements define the features of each component of the project that are either expected to have or a possible stretch goal. The functional requirements for the project are split into three components: user authentication, data visualization, and API functionality.

User Authentication

This section describes the functional requirements for a user to log in to this service.

Customer

The customer will be the organization who has paid for NOBL's service. Additionally, all users will be tied to an organization within the web application since an organization might want to have multiple users keeping track of their ad campaigns.

Default Log-In

The system must allow users to log into the web application using their email and a password.

Private Registration

The system must not allow the user to register on their own volition. For the customer to have an account on the web application, the user must directly request the creation of an account to an administrator by email. The administrator will then log into the Auth0 dashboard to generate a password and account for the user.

Selection of Ad Campaigns

The system must allow the user to select a campaign that the user wants to see data for. After logging in, the user will be prompted to select a campaign before proceeding to the dashboard.

Data Visualization

This section describes the functional requirements for the users to view their organization's ad campaign data results.

Graphical Data Results

The system must allow users to see their data to be shown on charts and graphs. This abstracts the data so that the users can understand what it actually means for the ad campaign data.

Export Data

The system must allow users to export a graph or data as part of a document or just an image (.png) file. When a graph or data is exported as part of a document, this means that the graph is rendered onto the document or in the case of the data, it will be rendered in an understandable form onto the document. Additionally, data can also be exported not just in .pdf form, but also in different formats such as .csv.

API Functionality

This section describes the functional requirements for the API. The API is required for data retrieval.

Retrieve Specific Data Fields

The system must allow users to retrieve specific data fields, not the entirety of a database table. For example, the user would be able to retrieve the nobl_score data field of the NOBL database for a specific page or the number of impressions for a page.

Connect to NOBL Database

The API must ensure a connection to the database without revealing the connection certification which would allow unauthorized access to the NOBL database.

Performance Requirements

Performance requirements are what allows the system to serve the users with adequate efficiency. If performance requirements are not met, the functionality of the system will be degraded and the user's experience will suffer as a result.

Data Visualization Rendering Speed

The rendering speed of the graphs and charts must be under 250 milliseconds to ensure that the user is not waiting too much to visualize the ad campaign data. With the challenges the team found in the Technology Feasibility document, the team concluded that it is imperative that 250 milliseconds since this is above average when it comes to the results of the time it took for all the data visualization libraries to render 50,000 data points. This is the main purpose of the project and thus, it is of utmost importance that the system renders the graphs and charts as fast as possible.

API Response Time

The data retrieval speed of the API must be at a maximum under 1 second. The purpose of this performance requirement is to ensure that the data will be there at a proper time to allow the graphs and charts to populate. If the API data retrieval process takes longer than 1 second, it will become a detriment to the user experience since this is the first step in rendering the graphs/charts.

API Error Rate

There are two kinds of errors that can occur in an HTTP request to a rest API. User error which is typically represented by the API returning a 400 level HTTP response code and server side errors which are typically represented by 500 level HTTP response code. The project must aim to keep both kinds of errors below a certain threshold as it would be a disruption to the service that the project provides to NOBL customers.

API Startup Speed

While downtime is undesirable, it is sometimes inevitable. In the case of downtime, it is important to be able to restore service quickly. Discounting the time needed for the server operating system to boot, the API startup sequence is the next most costly element in terms of downtime cost and thus, it is important to make sure that the API is able to quickly begin serving requests after a reboot.

4 - Architecture and Implementation

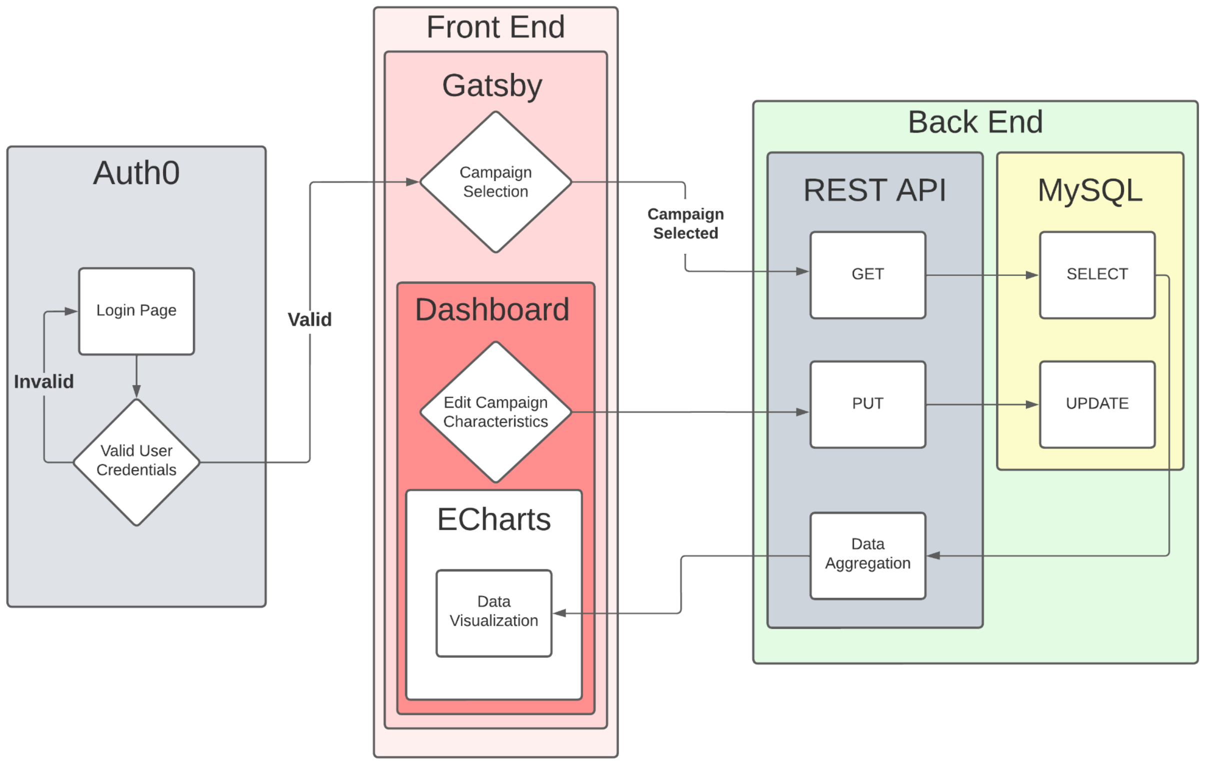


Figure 4.1 - Overview of the Project System

The system is divided into three parts: Auth0, a third-party component that handles authentication shown in Figure 4.1 in gray. Note the absence of an account creation mechanism because all account creations will take place via direct communication with NOBL.

After authenticating via Auth0, the user will reach the front-end web application (colored in shades of red in figure 4.1) powered by Gatsby.js, the framework that is responsible for quickly rendering the static elements of the webpage while ECharts is used to display the graphical depictions of NOBL's data. The two technologies work in tandem to produce the form and functionality of the web dashboard frontend and are expected to be the main interface to NOBL's data that NOBL's customers make use of.

In order to populate the web application with graphical data elements, data is queried from a MySQL database. The REST API translates HTTP requests into equivalent SQL queries, queries NOBL's backend MySQL database (denoted in darker-tan in Figure 4.1), aggregates the data, and returns the result in JSON notation for ECharts to process into graphical elements. The API's responsibility is not limited to the web application and may be used outside of it for organizations that wish to process the aggregated data themselves for their own reports or to display on their own websites.

This interaction between the web application and NOBL's backend MySQL database via REST API forms the core general loop of information flow within the project. The specific flow of information will be determined by the action the user takes when using the website.

Another thing to note about Figure 4.1 is that it does not depict the aforementioned technical users who may make use of direct access to the API and bypass the web dashboard. They can be thought of as existing in the frontend area given that they will still need to authenticate via Auth0 and query via the REST API effectively opting for a command line interface to the data as opposed to a graphical user interface.

There are a variety of components to this architecture design. It serves as an overview to a deeper description of software components. Each component will be described in this chapter.

Web Application

Three components are needed for the web application: a login page, a campaign selection page, and the dashboard page. The components will provide a familiar interface to NOBL client's. Aesthetics aside, the web application's functionality will allow NOBL clients to analyze and interact with their campaign data securely and effortlessly.

Login Page

The login page is the user's first module from the Misinformation and Credible News Analysis Tool they will interface with. NOBL Media provides clients with a username and password to login for their businesses account. Once the user enters their credentials, Auth0 will check if the user exists and if the passwords match. For security purposes, the login page will not tell users if the email is incorrect. The login page will present the user with a message saying the credentials were typed in wrong. If the login credentials are correct, the user will be sent to the campaign selector page.

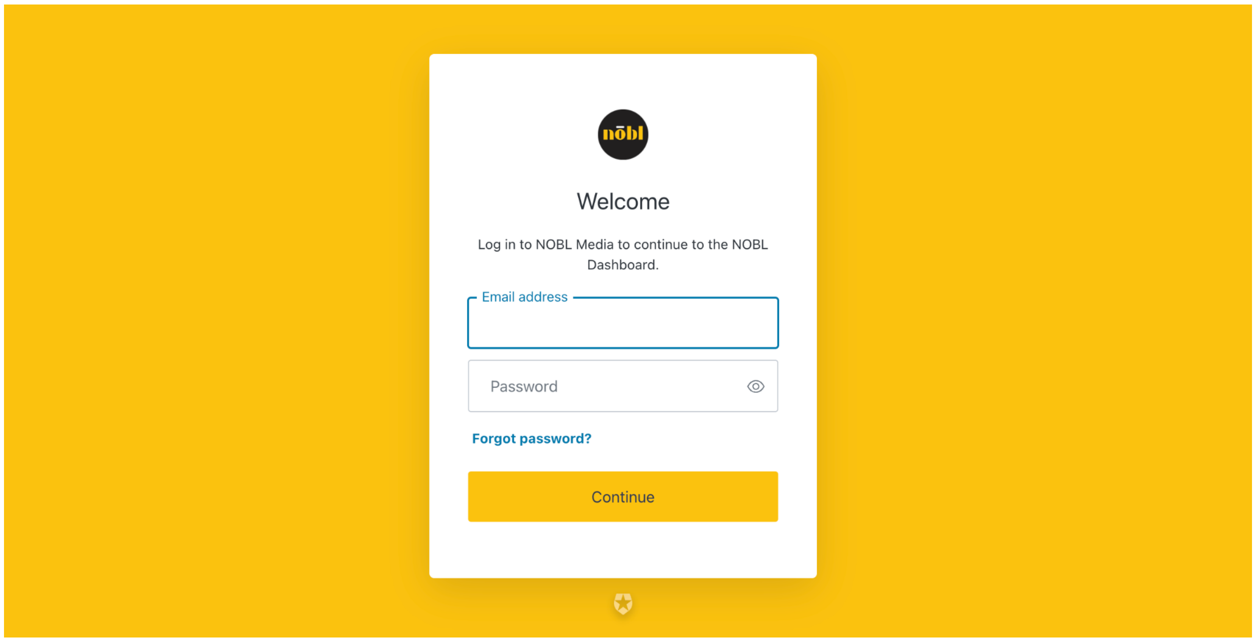


Figure 4.2 - Current implementation of the Login Page

Figure 4.2 shows the current team design for the login page. The user is prompted for their email address (username) and for their password. Should users find that they have forgotten their passwords, they will be able to reset their password by clicking the "Forgot Password?" button that is at the bottom in between the password input form and the continue button.

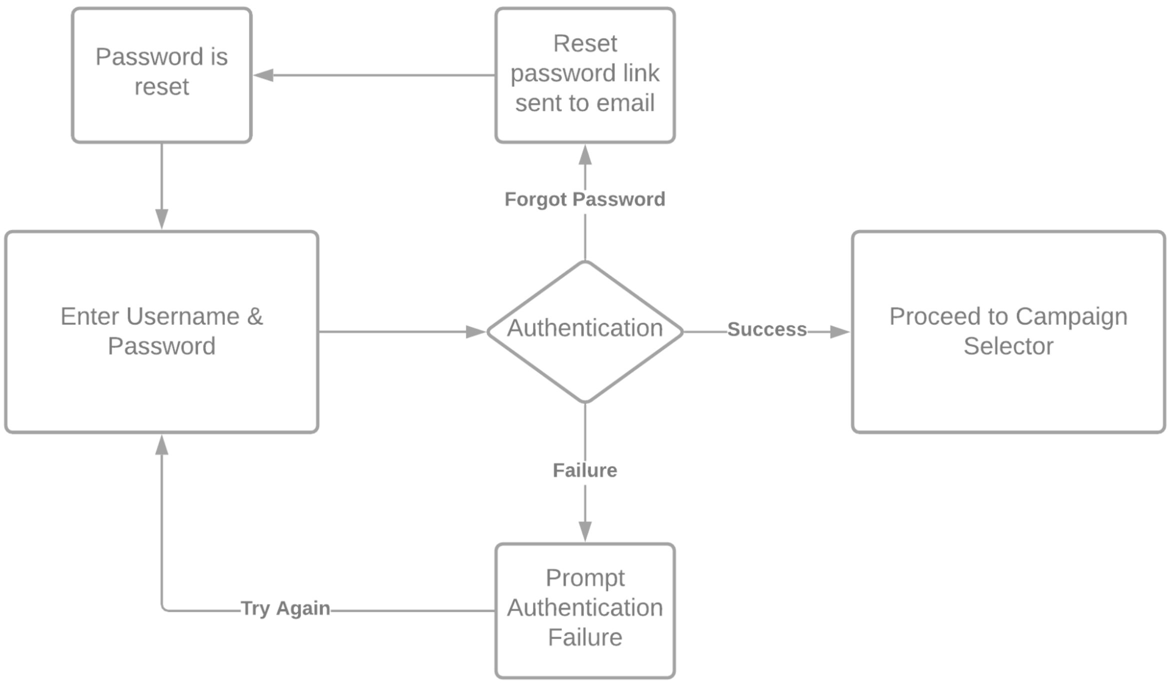
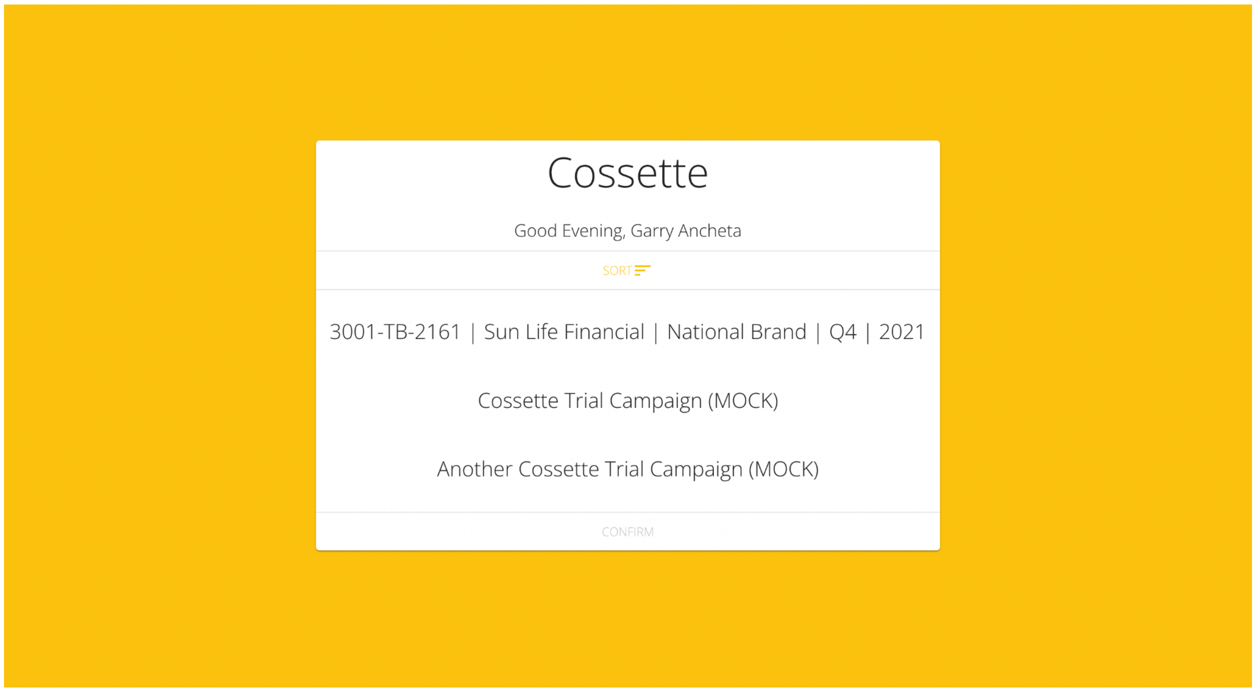

Password is reset
Reset password link sent to email
Forgot Password
Enter Username & Password
Authentication
Success
Proceed to Campaign Selector
Failure
Prompt Authentication Failure
Try Again

Figure 4.3 - Process diagram for the login page

Figure 4.3 shows a state diagram of the user process of logging in. It starts with the user entering their username and password. Should the user forget their password, they will have to click on the "Forgot password?" button and will be then sent a link to reset their password to their email. Once they have clicked the link and reset their password, they will then have to re-enter their username and password. Should the user enter their correct email address and password, authentication will proceed and check if the email address is found and the password for that specific email address matches the one inputted by the user. If both are found to be true, then the user is authenticated and can proceed to the campaign selector page. Should the user fail to authenticate themselves, they will be prompted with a notification that either their email address or password were wrong and will have to re-enter them once more.


Cossette

Good Evening, Garry Ancheta

3001-TB-2161 | Sun Life Financial | National Brand | Q4 | 2021

Cossette Trial Campaign (MOCK)

Another Cossette Trial Campaign (MOCK)

Figure 4.4 - Current Implementation of the Campaign Selector Page

Campaign Selector

NOBL Media clients can have multiple ad campaigns per account. At this point in the login process, a user will have to pick a single campaign to proceed to the user dashboard.

Referring to Figure 4.4, the header section of the page is where the organization name and the greeting of the day is located. Both the greeting of the day and the organization name is dynamically displayed, pulling data from the NOBL database to retrieve the user's first name and last name and the organization that the user is under. Additionally, the menu below the header section is also dynamically created and will only be fully initialized once the web application has retrieved all the campaigns that the user is associated with. Additionally, Figure 4.4 also shows the "Confirm" button, grayed out. This is controlled by the web application so that the button cannot be clicked by the user before clicking on a campaign. Once a campaign is selected, which the user will know since it will be highlighted on the menu, the confirm button will brighten and turn blue making it known that it can be clicked.

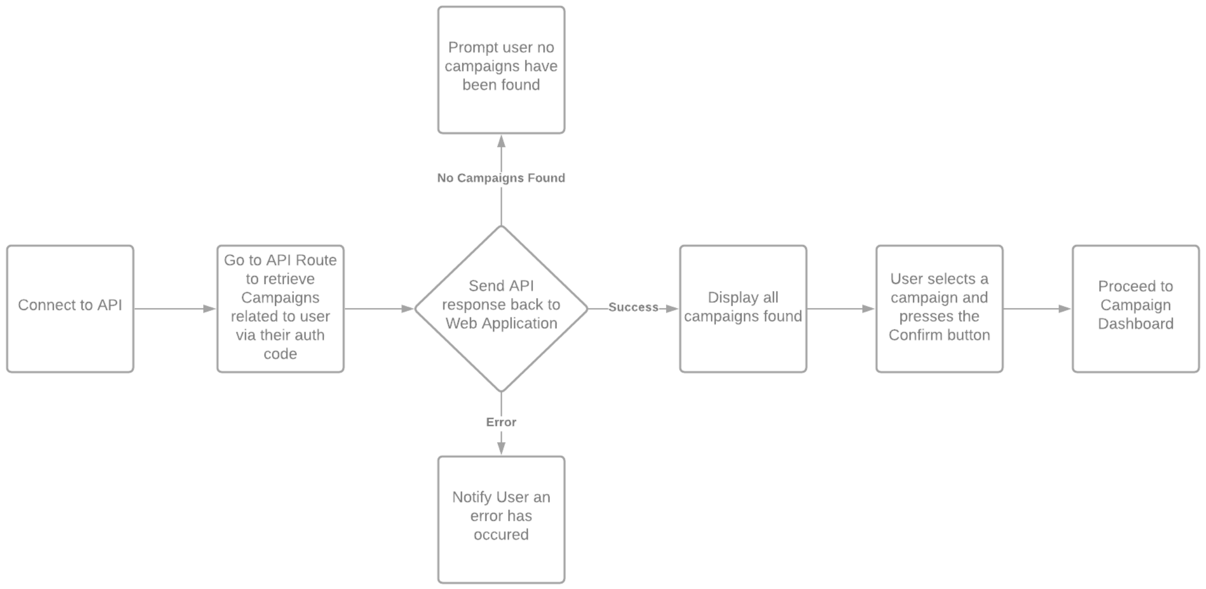

Prompt user no campaigns have been found
No Campaigns Found
Connect to API
Go to API Route to retrieve Campaigns related to user via their auth code
Send API response back to Web Application
Success
Display all campaigns found
User selects a campaign and presses the Confirm button
Proceed to Campaign Dashboard
Error
Notify User an error has occured

Figure 4.5 - User process diagram of the Campaign Selector page

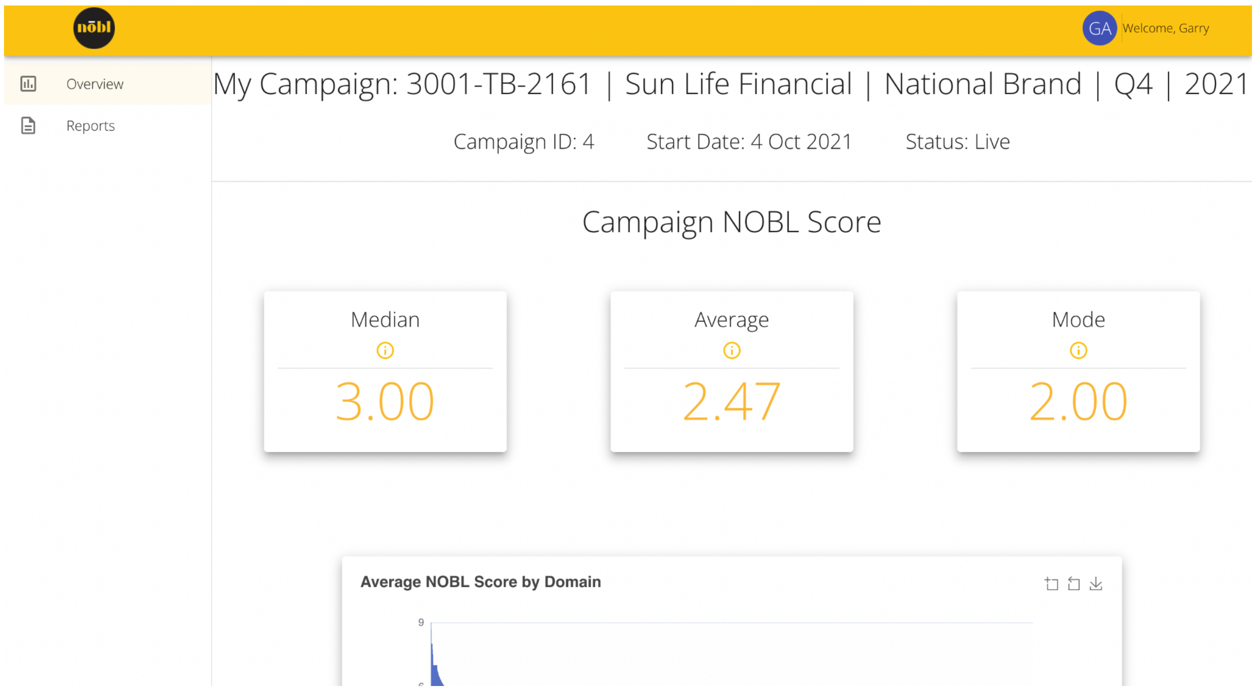
Figure 4.5 shows the basic workflow of the campaign selector page. Once the user has completed the authentication process referred to in Figure 4.3, as soon as the user is logged in, the web application will start the process at the beginning of Figure 4.5. In the underlying processes of the web application, the campaign selector page will connect to the API, which will then go to the REST API route for campaign data retrieval. The route will then attempt a query to the database to retrieve all campaign data that the user is associated with. This is where there are three possible results: an error, no campaigns found, or campaigns were found. If there is an error, this will be sent back to the web application and the user will be prompted with an error message. If there are no campaigns found, the menu section will prompt that no campaigns have been found for the user and should contact a NOBL administrator. If campaigns are found, the API will then send the information to the web application. The web application will then display the campaigns, allowing the user to pick one. Once the user has picked a campaign and has pressed the confirm button, they will be sent to the campaign dashboard.

Campaign Dashboard

After logging in and choosing an ad campaign users will finally arrive at the campaign dashboard page. This is the main page users will spend their time on.

The dashboard contains tabs for different campaign information. The main tab at the top allows the user to switch between their account’s ad campaigns. As seen on Figure 4.6 there are additional tabs showing different views of the campaign data. Figure 4.7 shows a close up view of the tabs.

- Overview
    - A summary of all campaign data. Provides a quick look at how the campaign is performing at this time.
- Reports
    - Allows users to retrieve specific campaign data in the form of a CSV file.



My Campaign: 3001-TB-2161 | Sun Life Financial | National Brand | Q4 | 2021

Campaign ID: 4 Start Date: 4 Oct 2021 Status: Live

Campaign NOBL Score

Median	Average	Mode
3.00	2.47	2.00

Average NOBL Score by Domain

Figure 4.6 - Current implementation of the Dashboard Page

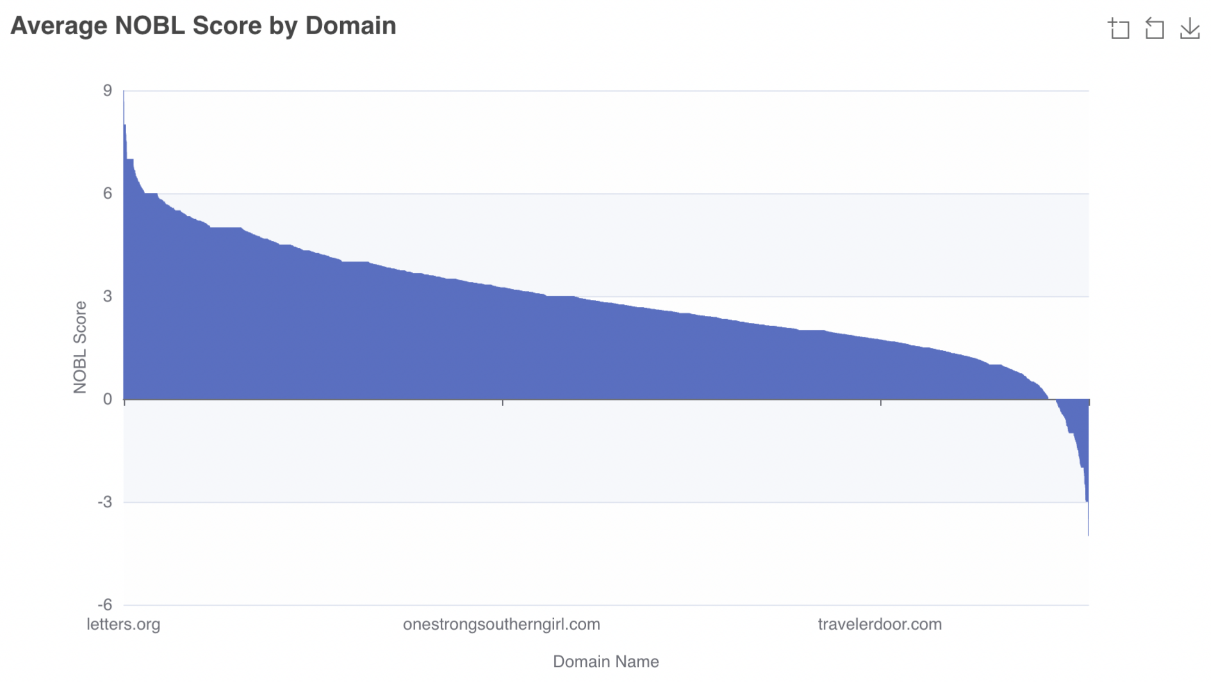


Figure 4.7 - Current implementation of a Graph

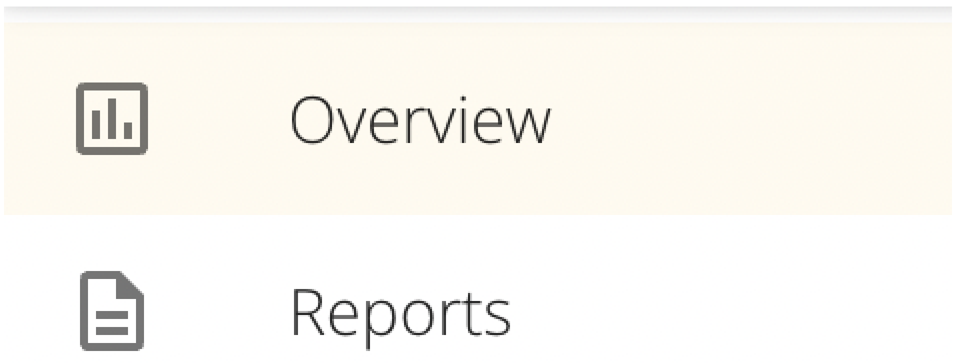
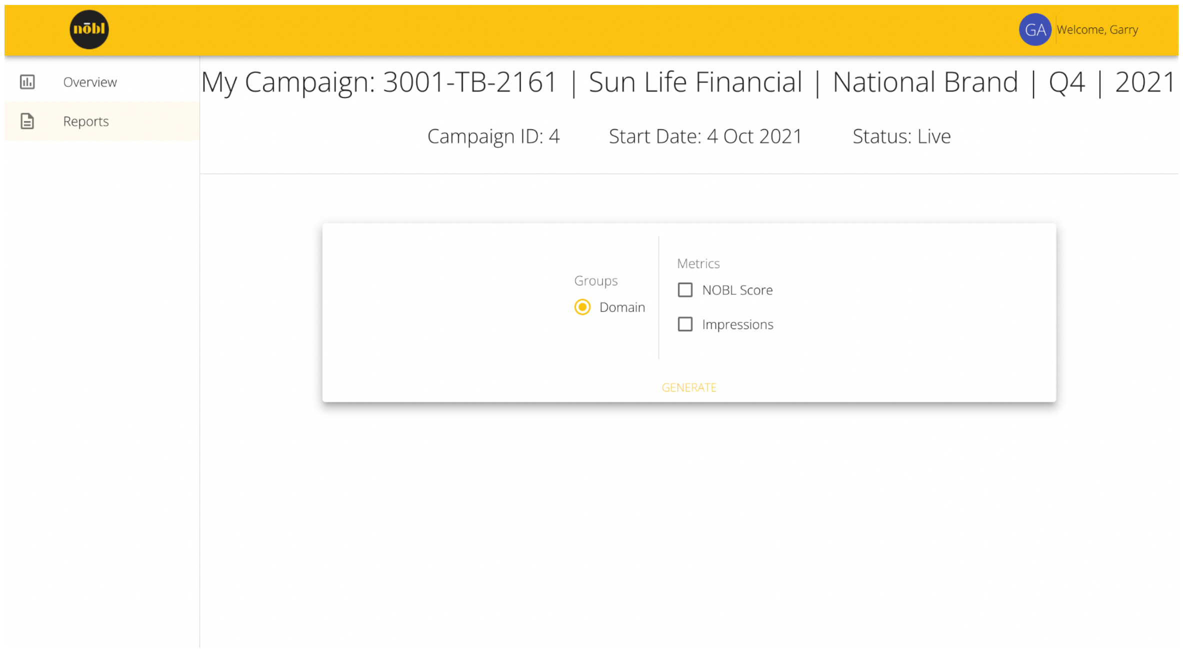


Figure 4.8 - Current implementation of the Dashboard Sidebar

As seen on Figure 4.6, there are different components to the entire Dashboard. On the left within Figure 4.6 is the sidebar which the user is able to use to navigate through different parts of the dashboard. Figure 4.8 is a deeper look at the side bar and the two tabs that are currently offered: the Overview tab and the Reports tab. At the top within Figure 4.6 is the NavBar, which allows the user to see their account icon and their name. The account icon can be clicked to open a menu which shows different settings such as changing the campaign and logging out. Immediately below the NavBar is the header which retrieves data about the current campaign being looked at using the API and displays the Campaign name, start date, status, and ID. Figure 4.7 shows a graph implementation which the user can interact with through zooming in and zooming out. Additionally, the graph itself can be downloaded as a picture in .png format for the user to use in different reports should they want to.


My Campaign: 3001-TB-2161 | Sun Life Financial | National Brand | Q4 | 2021

Campaign ID: 4   Start Date: 4 Oct 2021   Status: Live

Figure 4.9 - Current implementation of the Reports view

Figure 4.9 shows the current implementation of the Reports view. Here, the user can generate reports that are outputted in .csv format. The user can decide to get different metrics such as NOBL Score, impressions, or both. Clicking the "Generate" button will then pull data from the NOBL Database using the API and will automatically allow the browser to download the report.

5 - Testing

Unit Testing

Unit Testing is the process of testing small functional parts of the software to make sure that these parts are working as intended. To be more precise, Unit Testing is targeted towards functions within the software, which are the smallest "group" of code that is intended to output a desired result. Unit Testing can be seen as the first line of defense when it comes to prevention of bugs; which in the long run, prevents costly changes to the code base for NOBL Media, our client.

Additionally, Unit Testing is not just something that happens whenever a cycle of development finishes, it can be done as the development progresses or even before the development begins. The latter is what is known as Test-Driven Development (TDD) where the unit test is created first and then the functions are created in a way that it should pass the unit test. However, Team Truthseeker has not implemented TDD, but rather performed unit testing as the development progresses. Due to the nature of the project, being that it is split into the front-end user interface and the back-end application programming interface (API), unit testing can only be performed with the API. The API is the perfect environment to have unit testing because of the two following characteristics:

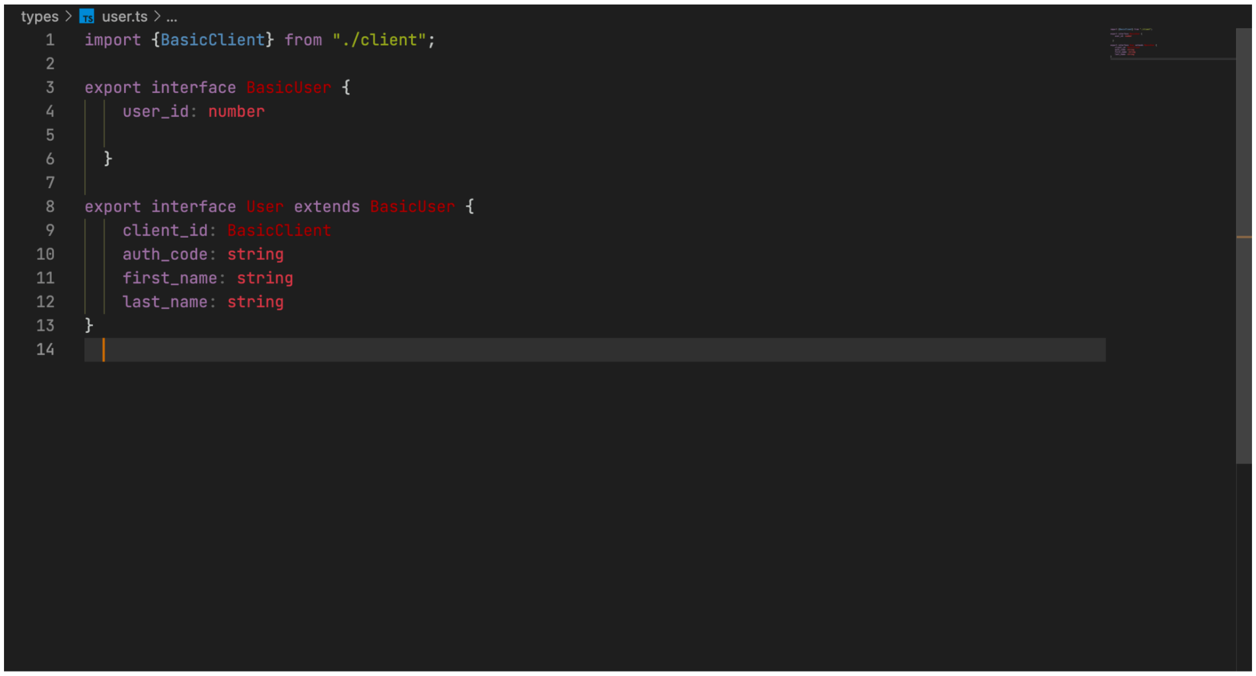
1. Small, modular functions
2. The API's purpose is to output data

The majority of APIs deal with pulling and inserting data from a source (usually a database) and then sending it to where the users are meant to see, manipulate, and create data. The NOBL API is designed to retrieve data from the NOBL MySQL database and take this data to display it onto the front-end. In terms of unit testing, it would be targeted towards the “simulation” of the front-end asking the API for data. The API that Team Truthseeker has built is, unintentionally, designed so that it is perfect for unit testing. Therefore, Team Truthseeker does not need to modify the API so that it fits unit testing, instead, the team has been able to directly go straight to unit testing. Furthermore, the UI (user interface), which is the front-end of the project, does not need unit testing since UI testing is more complex and cannot be broken down to simple components like how unit tests should be.

To make unit testing the API easier, Team Truthseeker will be using AVA which streamlines the process. AVA is a minimalistic unit testing framework which skips over the need to create unit testing functions. One way of performing unit tests is called matching, where parameters are set for a certain function whose output will be *matched* with an expected output. If the function’s output does not match the expected output, then the unit test fails, and if it passes, the unit test passes. An example of this type of unit testing is if there was a function which performs addition. The unit test would put in different numbers (ex. 4 and 5) and then would match this with the expected output (ex. 9). If the function outputs 5, then the unit test fails, which then means that there is a bug in the function.

The NOBL API is is structured into three main components:

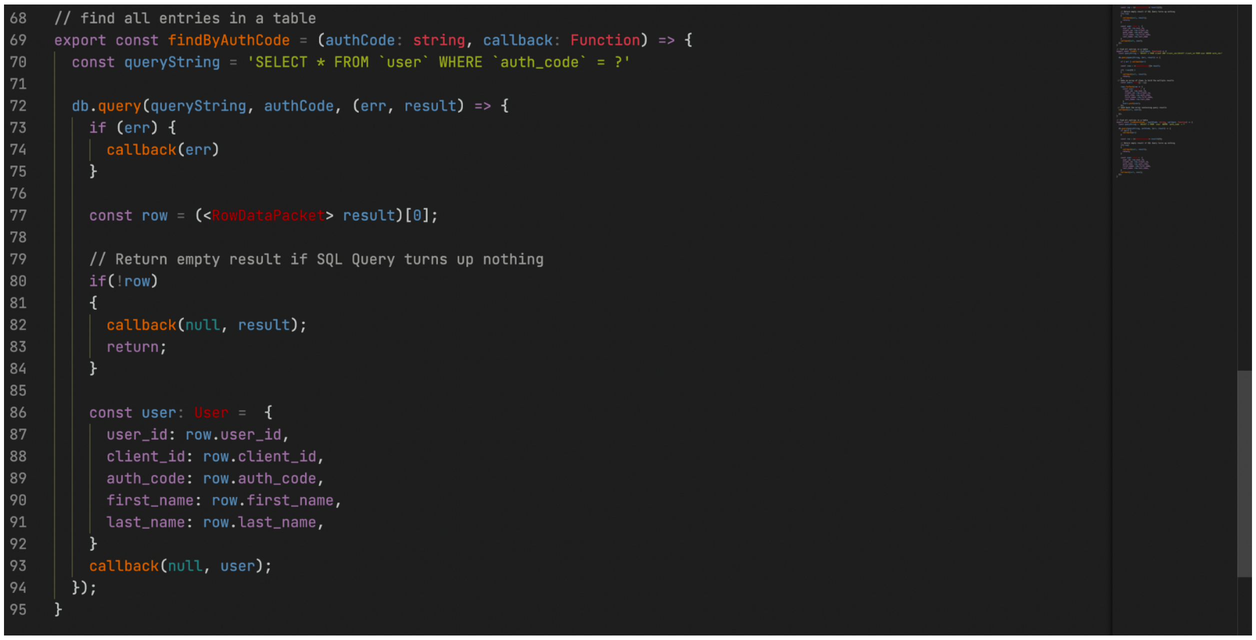
1. Routes
2. Models
3. Types


types > user.ts > ...

```
import {BasicClient} from "./client";

export interface BasicUser {
    user_id: number

  }

export interface User extends BasicUser {
    client_id: BasicClient
    auth_code: string
    first_name: string
    last_name: string
}

```

Figure 5.1 - A User Type

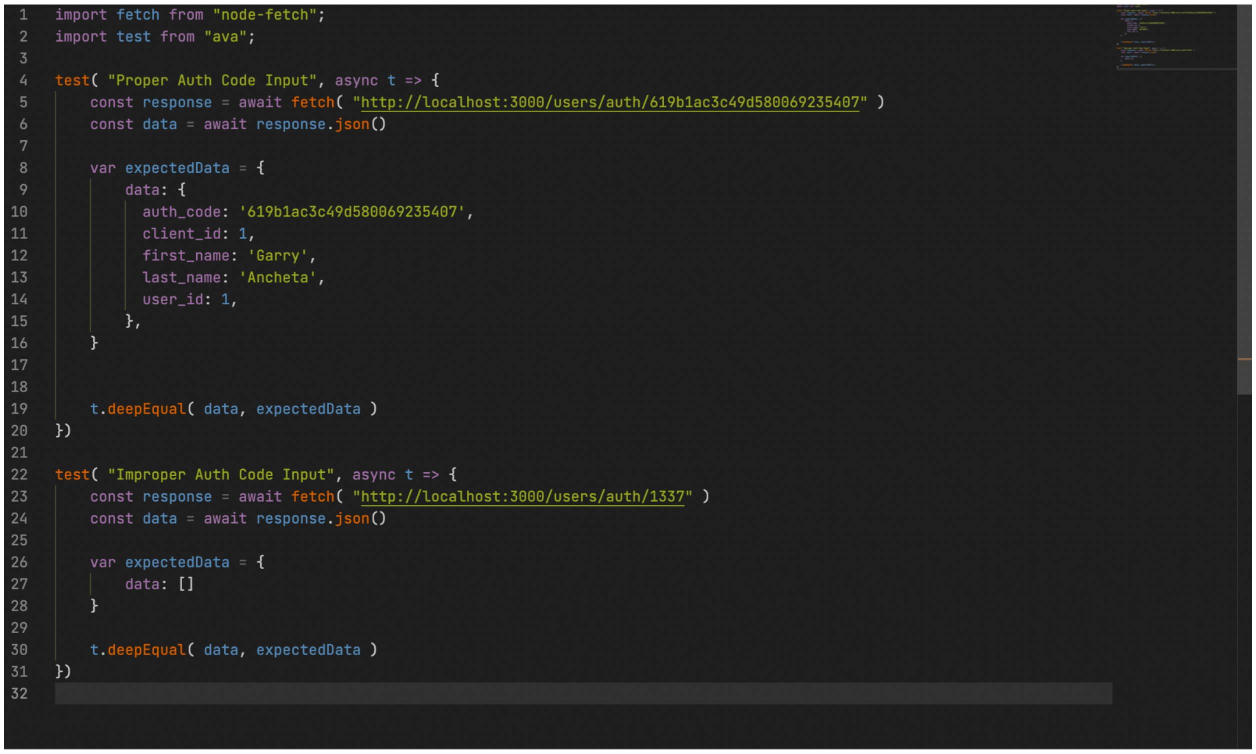
The Routes component allows for the front-end to navigate through the API, allowing it to retrieve specific data that it needs, not just everything the API provides. The Types component allows the API to define what it expects the data to be when it pulls it from the NOBL database. Referring to Figure 5.1, this is an example of a type, this allows the Team to omit unit testing between the API and the NOBL database because when the API pulls data from the NOBL Database, it matches the data to the type first to make sure that the data is actually what is intended and remove any other data. So in the case of Figure 5.1, if the API was to pull data for a user, it would ask the database for the data and then check the data that was sent with the type. So, as seen in Figure 5.1, if there was a birth_data field that was sent to the API by the NOBL database when being pulled, the API will just throw that data away since it is not needed.



```
// find all entries in a table
export const findByAuthCode = (authCode: string, callback: Function) => {
  const queryString = 'SELECT * FROM `user` WHERE `auth_code` = ?'

  db.query(queryString, authCode, (err, result) => {
    if (err) {
      callback(err)
    }

    const row = (<RowDataPacket> result)[0];

    // Return empty result if SQL Query turns up nothing
    if(!row)
    {
      callback(null, result);
      return;
    }

    const user: User =  {
      user_id: row.user_id,
      client_id: row.client_id,
      auth_code: row.auth_code,
      first_name: row.first_name,
      last_name: row.last_name,
    }
    callback(null, user);
  });
}
```

Figure 5.2 - One Part of the User Model

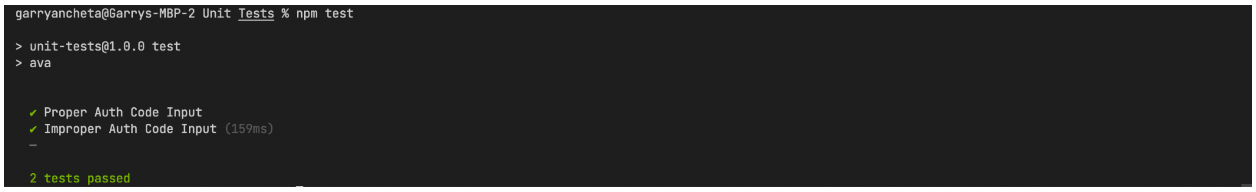
The Models component is where unit testing comes in. The Models component contains the functions that retrieve data from the database. In this case, the unit tests are for when requests from the web application are sent to the API and are waiting for a response. Referring to Figure 5.2, this function is ready for unit testing since it is possible that there might be an underlying bug or an improper error handling that can be prevented.



```
import fetch from "node-fetch";
import test from "ava";

test( "Proper Auth Code Input", async t => {
    const response = await fetch( "http://localhost:3000/users/auth/619b1ac3c49d580069235407" )
    const data = await response.json()

    var expectedData = {
        data: {
          auth_code: '619b1ac3c49d580069235407',
          client_id: 1,
          first_name: 'Garry',
          last_name: 'Ancheta',
          user_id: 1,
        },
    }


    t.deepEqual( data, expectedData )
})

test( "Improper Auth Code Input", async t => {
    const response = await fetch( "http://localhost:3000/users/auth/1337" )
    const data = await response.json()

    var expectedData = {
        data: []
    }

    t.deepEqual( data, expectedData )
})
```

Figure 5.3 - Snippet of Unit Testing Code

Figure 5.3 shows a code snippet of a unit test for the NOBL API. There are two tests within the picture shown, one for when a proper input is provided, and when an improper one is not. In both tests, data is being actively pulled from the API endpoint (a URL that the API has set up from which the web application or a unit test can grab what the API is grabbing). The test has a variable for expected data, the variable "expectedData" which it uses to compare the response from the API endpoint. It then uses the deepEqual function to compare the data grabbed from the endpoint and the expected data. Should they be equal, then the test passes.



```
garryancheta@Garrys-MBP-2 Unit Tests % npm test

> unit-tests@1.0.0 test
> ava


  ✔ Proper Auth Code Input
  ✔ Improper Auth Code Input (159ms)
  ─

  2 tests passed
```

Figure 5.4 - Results of the Unit Test

Figure 5.4 shows the result of the Unit Test snippet. In this case, both tests pass and therefore, it can be concluded that the API endpoint for users provides the proper responses for both using a proper input as well as an improper input.

Within the coming weeks, Team Truthseeker intends to actively perform unit tests as the API changes. By the end of the project, the unit tests should be comprehensive enough to cover all API routes.

Integration Testing

While unit testing is important for ensuring intra-module quality assurance by verifying the expected functionality of functions execution, modern software applications are complex multi-module systems which often have separate teams working on each module with limited communication between them. This has the potential to degrade the cohesion of the software product and in extreme cases cause significant development delays in otherwise well managed projects. This is where integration testing complements unit testing. Integration testing is the process of ensuring that software modules integrate in the expected way during software usage. This can be thought of as the inter-module counterpart to the work done via unit testing.

To further illustrate this distinction, consider the following metaphor: if unit testing is doing quality control checks on car parts at the factory, integration testing is the process of taking the car out on the test track and making sure the brake pedal module integrates with the wheel module and stops the car as expected when the two components are used together and that unrelated systems do not affect each other such as making sure that turning on the radio does not turn on cruise control or vice versa.

While the team is quite small and in communication about the work the team is doing on the software modules the general principle of integration testing is still quite important to the project given the projects two modules together result in a minimum of 4 changes in technical context for any given user interaction. These 4 changes in technical context during execution of program functionality form the basis for the testing plan.

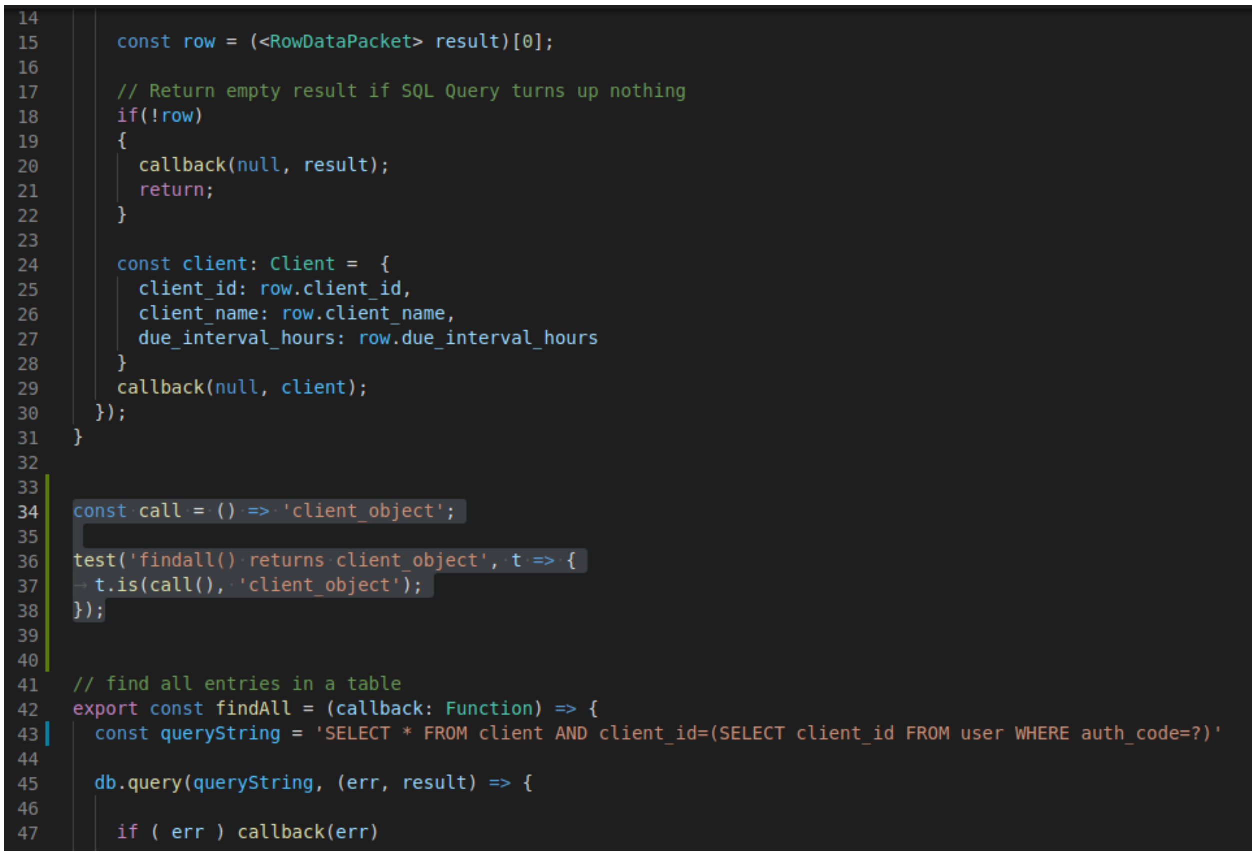
The web application begins by taking client HTTP requests to Auth0’s third party authentication server from which is passed an authentication token which logs the user in and displays their information. Here lies the first challenge of testing for proper integration of the third party authentication system with the first party website software. This particular technical context switch is especially important because failure to authenticate properly risks allowing access to data from both NOBL media and their clients.

Thankfully, easing the difficulty of testing this section is Auth0’s well documented ready made libraries designed for integration in small projects such as the web application and its associated API. One of the functionalities included in this library is error generation if the authentication process fails. This means that barring some implementation specific mistake in the codebase this context switch from third party authentication to first party website content should be largely seamless and any failures that occur should be highly visible during usability testing and related activities meaning that little if any integration testing specific code needs to be written by the team to cover this case.

The next change in technical context to be considered in when the web application once the user is logged in queries API data via HTTP requests which are translated into SQL queries and executed against NOBL Media's backend MySQL database. This is arguably the most complex and error prone context switch because it involves not only the translation of HTTP requests into SQL queries via the API's SQL query templating engine but is also responsible for passing the Auth0 authentication state from the website to the API to allow the API to query only data related to the currently logged in user.

The test for this using the AVA framework will be run using a variety of both correct and malformed requests to check that SQL queries are produced as expected for correctly defined requests and that the system fails gracefully and returns an error rather than passing a malformed query with undefined behavior to the MySQL Backend.

Following along this code flow is the next step of testing whether or not the SQL queries return the expected results from NOBL Media's backend MySQL server. Given that the functionality of this component is largely dependent on NOBL Media's database architecture the only real non blackbox component to be tested is whether or not the API fails gracefully if the database is not available or a malformed SQL query is passed to the backend MySQL server and returned to the API.



```

    const row = (<RowDataPacket> result)[0];

    // Return empty result if SQL Query turns up nothing
    if(!row)
    {
      callback(null, result);
      return;
    }

    const client: Client =  {
      client_id: row.client_id,
      client_name: row.client_name,
      due_interval_hours: row.due_interval_hours
    }
    callback(null, client);
  });
}


const call = () => 'client_object';

test('findall() returns client_object', t => {
  t.is(call(), 'client_object');
});


// find all entries in a table
export const findAll = (callback: Function) => {
  const queryString = 'SELECT * FROM client AND client_id=(SELECT client_id FROM user WHERE auth_code=?)'

  db.query(queryString, (err, result) => {

    if ( err ) callback(err)
```

Figure 5.5 - Results of the Unit Test

This section is tricky to test because the output of a query can change when the backend database changes making reproducible results difficult for certain kinds of queries. Because of this it makes the most sense to test for the structure of the data being transferred correctly more than the data itself.

As seen In pursuit of this goal this section of the codebase already has implicit data structure validation and early callback exit on error through the codebase's use of Typescript which enforces object and variable structure through its static typing capabilities. Testing is only needed for edge cases such as queries that are valid but return empty results or queries that return very large amounts of data.

The final context change the data goes through is that it is converted into JSON notation before being returned to the user as an HTTP response. This is arguably the most straightforward conversion and is done in literally one library call so the procedure here is much the same as previously where we verify the JSONification library call is resulting in the expected output for a standardized set of Typescript object inputs. After the response object is JSONified it is simply returned to the website which reads and displays the data graphically

If we implement these tests at each context change that occurs during normal program execution it should support the existing mitigation measures in ensuring a high quality codebase where anomalous behavior stemming from module interaction is discovered and prevented prior to deployment of our software in a production environment which should reduce both the number of customer complaints and increase code maintainability in the long run.

Usability Testing

With the integration testing having been completed, the last test is the usability test which is described within this section of the document. Also referred to as user testing, usability testing measures the overall user experience on a product. Specifically with this tool, the test is to assess how user-friendly and functional this web application is. In the planning stages of this test, a selected number of users were chosen to test out various functions throughout the web application. In doing so, the users described their experience and answered a series of questions.

In testing end-users on the Misinformation and Credible Analysis Tool, at least 6 users were selected. Since NOBL Media's clients were not accessible for testing, the criteria of the chosen testers needed to be aligned with NOBL Media's clientele, who are the intended users. NOBL Media is a company that mainly services companies that do any sort of advertising; the audience they reach out to are employers of these companies who work in the marketing and advertising division. This means the end-users needed to simulate a similar background. So, the team sent out an email to 20 users of similar backgrounds in order to obtain at least 6 users to test this software product. This number of users was enough to give adequate feedback, but not too much to dilute the results.

This test used techniques to ultimately gather qualitative data, where the team recorded and analyzed user interactions with this product. About 6 users were needed to set up pair testing, which was used to compare specific user interactions. Each pair test needed to be set up similarly in order to keep the results aligned with one another, so the team created a script asking the users questions over their interactions and overall experience. This also helped guide the usability test to ensure the users interact with specific functionalities that needed to be tested and assessed.

Diving deeper into the specifics of the user testing for this web application, each pair test was asked a set of questions guiding them to cover the following functionalities:

- User invitation
- Failed log-in
- Successful log-in
- Select Campaign
- Switch between tabs on the dashboard
- Download data
- Logout

Once the team added the user's emails to the authentication system, the test began with user invitation. After this, each user attempted to log into the web application twice; once with an incorrect email or password and once with a correct email and password. Once the user attempted the failed log-in, they were prompted to select the button labeled as “forgot password” to ensure they were redirected to a new page where they could input their email. Before attempting a log-in with the correct credentials, the user checked to ensure they received an email to reset their password and whether resetting the password was successful. Following this, the user had to go back to the web application to successfully log-in so they could be directed to the campaign selector. Here the user was able to view all campaigns associated with their account and click on whichever campaign they desire. From here the user was directed to interact with the overall dashboard: switching between tabs, refreshing the page to see accurate data, and being able to download this data. Once the users interacted with the dashboard in these areas, they were prompted to select the tab to log out of their account.

After the user testing sessions were complete, the team gathered all collected data to come to a conclusion over the usability for this product. The overall consensus was that this product was very straightforward and easy to use. Users had very little to no complications in performing each task without much direction from the team. However, a bit of guidance seemed to be needed in regards to the dashboard metrics as most users were not too familiar with things such as the difference between average and median.

6 - Project Timeline

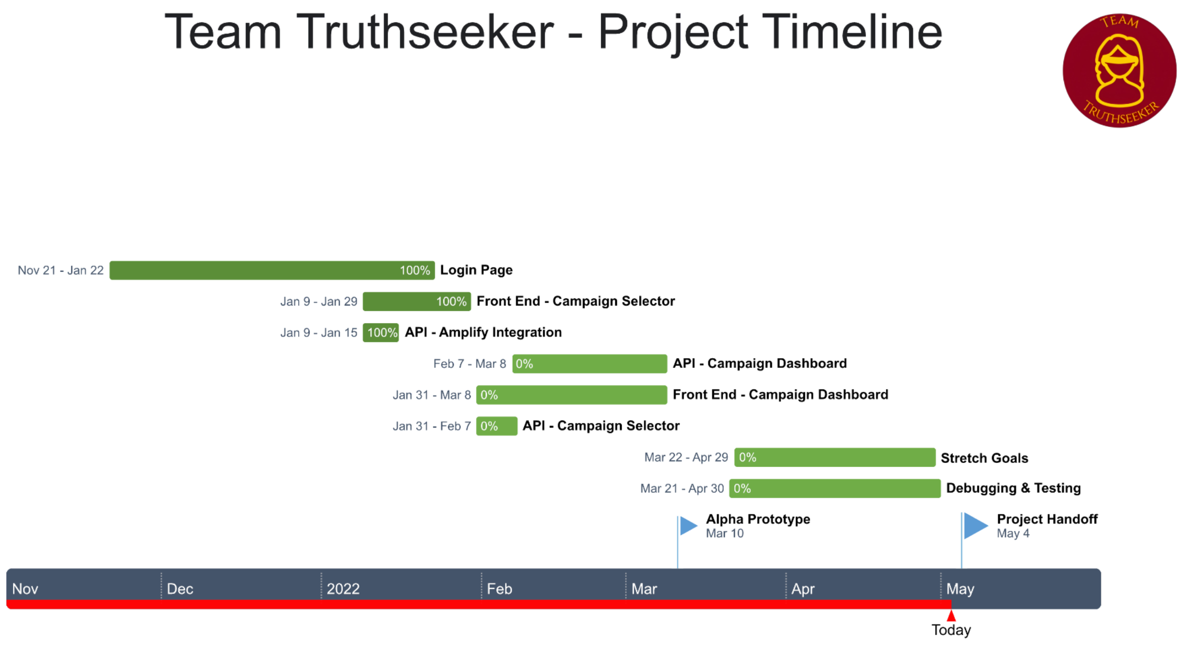


Figure 6.1- Final Project Timeline

Our project timeline consists of the start of the team's implementation of the project on 21 November up until the Project Handoff that is scheduled for May 4th. Once the Login Page was set up, at the beginning of the Spring 2022 semester was when the team started to hike up the implementation process and was able to implement the Campaign Selector and integrate the current API into Amplify, an Amazon cloud hosting service. Once the integration and Campaign Selector was complete, the team moved on to implementing the Dashboard and the backend functions for the Campaign Selector. The dashboard took the longest to complete, but the team was able to accomplish a full Alpha prototype by 10 March. After the Alpha Prototype and Spring Break, the team moved on to stretch goal implementation as well as debugging &

testing. Finally, the team is looking forward to meeting with our client to officially transfer all information and data.

Throughout the entire implementation timeline, the team was split up into two groups: one responsible for the web application and the other responsible for the API. Additionally, the most complicated part of the project was not actually any implementation of the different components needed, but rather the usage of Amplify with the project. Amplify brought a lot of complications even up to the end when the team was finalizing the project. However, the benefits of using Amplify far outweighs the complications that the team has faced.

7 - Future Work

This section talks about some aspects of the project that the team believes can be extremely beneficial to the service the project provides or extremely useful in optimizing the web application and the API.

Campaign Selection Sorting/Filtering

In the case of the Campaign Selection, there is a simple sorting feature that has been implemented by the team. However, this sorting feature only allows for the alphabetically ascending/descending order of the campaigns as well as an ascending order of the dates the campaigns were created. Further improvements on this feature should be made to make it extensive enough to satisfy most user use cases.

Campaign Comparisons

This was something that was discussed by the client because of its importance to showing NOBL Media's service. Being able to go to the dashboard and start comparing the current campaign selected to a different campaign can show more benefits of NOBL Media's service. While this was something that the team had entertained to implement, the plan was scrapped due to time constraints.

Real Time Analysis

Currently in the dashboard, there are tooltips that can be hovered upon to learn more about to learn more about a specific metric. However, the team has thought of somehow providing real-time feedback towards the user about what the metric actually means towards their campaign. For example, a user would be able to hover over the NOBL Score tooltip and then instead of just defining the NOBL Score, the tooltip would instead give insight on what this means for the campaign as a whole; i.e. "The NOBL Score for the campaign shows that the advertisements appeared in extremely un-credible web pages. This can be changed through increasing the NOBL Score threshold. "

Live Campaigns

This feature would tremendously help NOBL Media provide a better service whenever an ad campaign is still being run. NOBL Media's database is updated consistently currently, it is just the web application that needs to be modified to ensure that data is actively being pulled.

8 - Conclusion

Due to the lack of proof that can be shown to NOBL Media customers, NOBL Media cannot truly show that their service has any benefits. The project has two components, the web application and the API. These two components work together to achieve the solution to NOBL Media's problem: show how NOBL's service actually benefits their customers.

To provide a good solution for NOBL Media's problem, the two components mentioned above have the following features:

1. The web application must be able to authenticate users to allow secure access to the customer's data using integrated technology Auth0.

2. The API must be able to handle user authentication requests.

3. The web application must be able to abstract JSON data to represent these to customers through graphs and charts.

4. The API must be able to retrieve the JSON data from the NOBL Media MySQL database.

5. The web application must allow customers to download customer ad data in a formatted file such as in a CSV or Excel file.

The project has been fully realized and the team has implemented some stretch goals to further provide a better experience for the user. Due to the completion of the project, NOBL Media can now integrate the project with their service to provide a good example of what NOBL Media's service can provide to their customers' advertisements. The project at its completion calculates 5 different data metrics and renders 2 graphs with more than 1000 data points.

Over the course of the academic year, Team Truthseeker has had their fair share of stressful moments but enjoyable moments as well. Luckily, all of the project's technologies were of great use and there were no extreme troubles regarding any implementation of the technologies the project used. Additionally, the project's process was extremely straightforward thanks to Team Truthseeker's careful planning. Overall, Team Truthseeker is extremely satisfied with the project's outcome and even wanted to work more in providing NOBL Media with the best possible state of the product.

9 - Glossary

This is a Glossary for different definitions that need to be defined to ensure understanding for the reader of different terminologies.

Repo - Short for "repository"; repo refers to the specific storage location of data; in the case of the project, this would refer to the location of the project's components.

CLI - Stands for "Command Line Interface"; this refers to any packages that are specifically for a service such as Amazon AWS, that is used using the computer terminal.

Fork - to create a new copy of a repo disassociated entirely from the original repo. This forked repo can be merged into the original copy as well by way of a pull request.

Appendix A:

Development Environment and Toolchain

This appendix will talk about the different tools that have been used in the implementation of the project. Additionally, this appending will also walk through how to set up the project itself and the production cycle used to progress the project.

Hardware

The project, composed of the API and the Web Application, was run on two platforms: MacOS and Windows. There was an attempt to run the project using Linux, however, only the API was successfully run. However, due to the similarities with MacOS and Linux as they are both Unix-based, the team believes that the web application can be run when using Linux. The MacOS was run in a Macbook with an M1 Chip bought just last year and had no problems with running either components. There were two devices that ran both the API and the Web Application and used Windows. The first is a desktop that is fairly new, bought within the past year. The desktop was able to run the API and the Web Application without any problems. The second is a laptop that is much older; this laptop was able to run the API successfully, but had trouble running the Web Application at the start of the implementation of the project. This was eventually resolved through reinstallation of the Web Application development environment. The Linux laptop was also old and as of this writing, still had problems running the web application completely.

The errors that occurred are usually pertaining to the packages that need to be installed for the project. Team members were able to solve some package incompatibility issues, but had not run the web application successfully.

Conclusion:

The project can be run under Linux, MacOS, and Windows operating systems. A newer model of device is ideal for the web application to be run, though it can be run on older models, but be aware of errors that might occur.

Tools

Here, the tools that were used to create and implement the project will be discussed.

Microsoft Visual Studio Code

Microsoft Visual Studio Code was the source code editor that was used by the team to implement the project. No additional extensions were needed within Visual Studio Code to implement the project, though some stylistic themes can be added to make the software look and feel better when used. Visual Studio Code provided a simple way for the team members to look at the code and contribute to the project.

Github / Github Desktop

Github was the git repository the team used for version and quality control for the project. Github Desktop further simplifies the process of creating, pushing, pulling, and committing code into the project. With Github Desktop, cloning the project is extremely simplified in comparison to using the command line as well. The team used Github Desktop to streamline the process of pushing and pulling code that was being contributed to the project; additionally, Github Desktop was useful whenever the project needed to be transferred to a new device without the hassle of ensuring that the project was cloned properly. Additionally, Github Desktop works directly with Visual Studio Code, and thus it makes contributing to the project easier.

Setup

Here, setting up the development environment for the web application and the API.

API

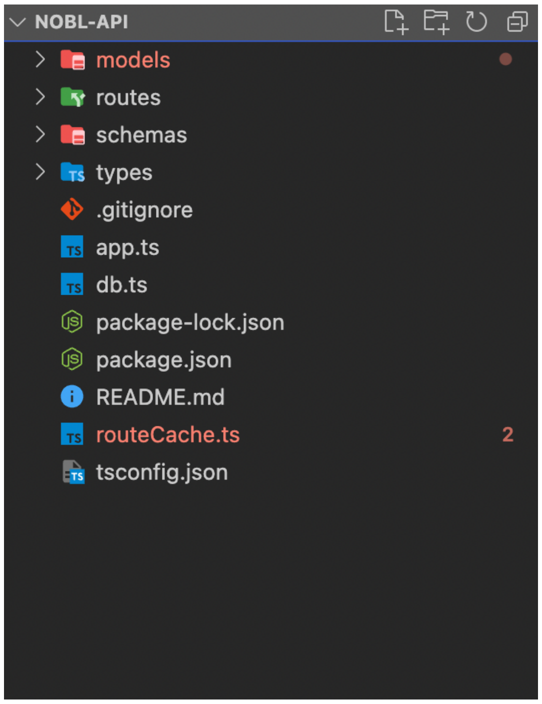
1. Clone the Github Repository into your computer.
    a. Using Github Desktop:
        i. Once Github Desktop is opened, on the top left corner of the software, click the Current Repository button to bring up the dropdown menu.
        ii. At the top of the dropdown menu, click the "Add" button.
        iii. A menu will appear, click Clone Repository.
            1. **NOTE**: In this menu, the location of where the repo will be stored in the computer can be set at the very bottom of the popup.
        iv. Scroll to find the repo, named "NOBL-API" and click on it.
        v. The API should now be stored locally.

NOTE: Github Desktop has functionalities to easily open the project with Visual Studio Code; it is recommended to use Visual Studio Code to contribute to the project.

b. Clone using the command line using Git or the Github CLI

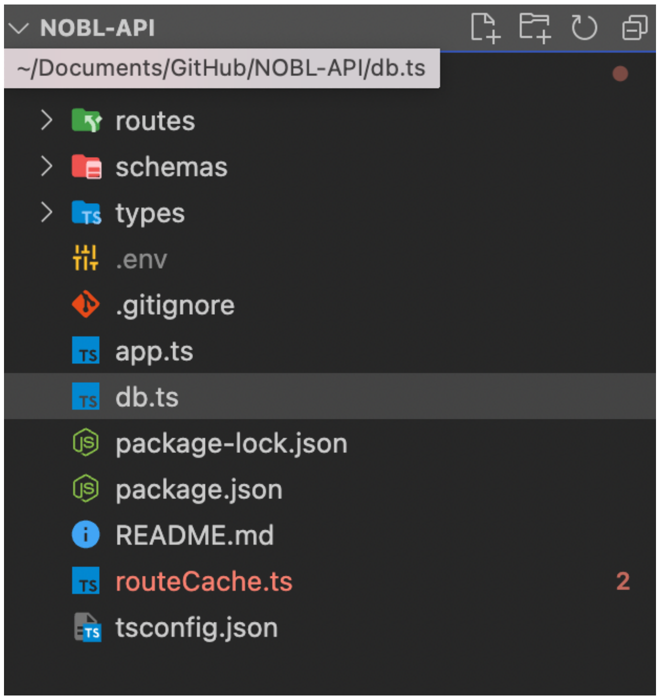
2. Open the repo in an IDE of your choice.
    a. **NOTE**: it is recommended to use Visual Studio Code

3. Confirm that the repo looks like the following picture:



NOBL-API
- models
- routes
- schemas
- types
- .gitignore
- app.ts
- db.ts
- package-lock.json
- package.json
- README.md
- routeCache.ts 2
- tsconfig.json

4. In the main directory, create a new file and name it ".env". The repo should now look like the following picture:



5. Open the ".env" file and ensure that the each variable is in the file as such:

```
PORT=[redacted]
DB_HOST="[redacted]"
DB_USER="[redacted]"
DB_PWD="[redacted]"
DB_NAME="[redacted]"
DB_DIALECT="mysql"
CACHE_DURATION=30
```

   a. For the PORT, DB_HOST, DB_PWD, and DB_NAME, ensure that you insert the credentials that are needed to connect to the NOBL

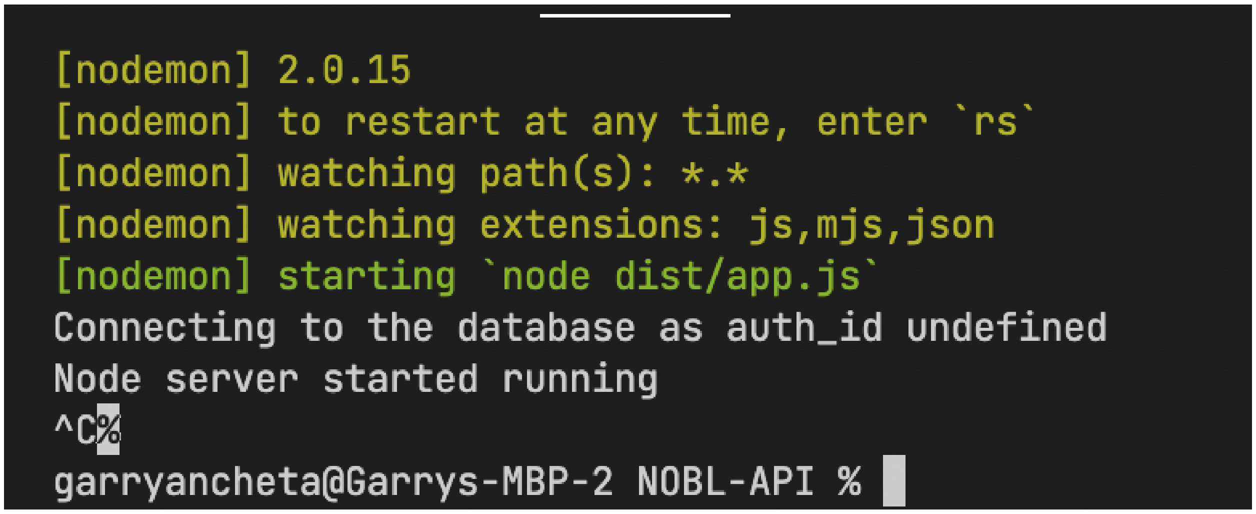
Media Database. Otherwise, the API cannot connect to the database and will not run properly.

6. Open up a terminal in your IDE.

    a. **Visual Studio Code ONLY**: On the toolbar at the top, click on Terminal and then new Terminal.

7. Within the terminal, enter "**npm install**" on the command line, and run the command. This will then install all the necessary packages that the repo needs which can be seen being installed in the terminal. At the end, a new folder should have been created named "node_modules."

8. Finally, on the command line in the terminal, enter "**npm start**". This will try to launch the API locally and if everything works, you can visit the link "localhost:[PORT]", where PORT is the port in the ".env" file and you will then see the following on the terminal:



```
[nodemon] 2.0.15
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node dist/app.js`
Connecting to the database as auth_id undefined
Node server started running
^C%
garryancheta@Garrys-MBP-2 NOBL-API %
```

Web Application

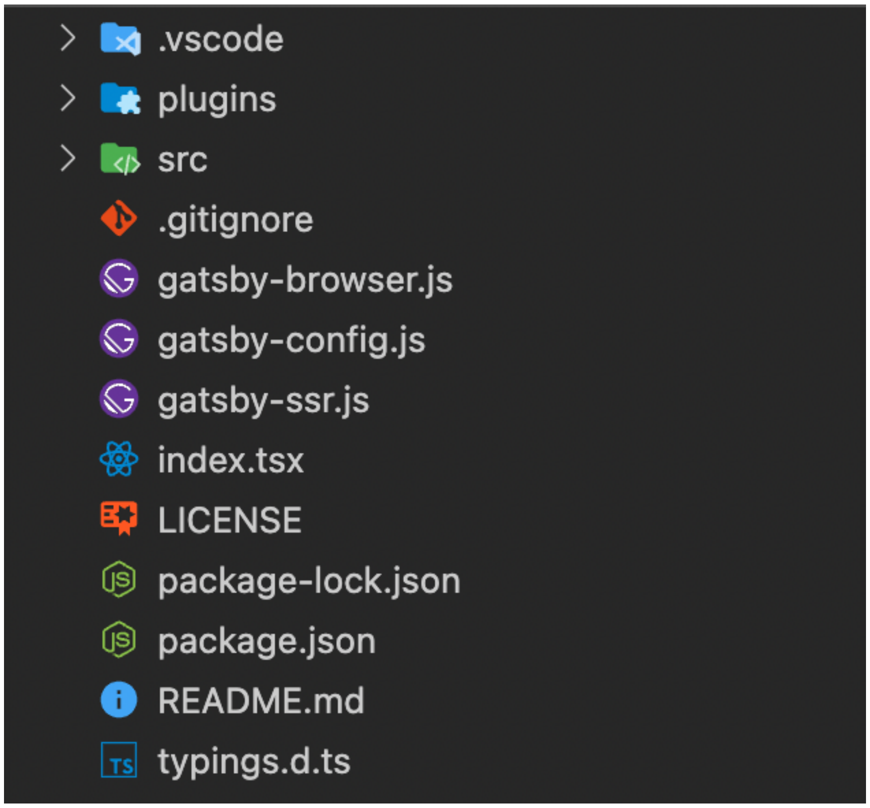
1. Clone the Github Repository into your computer.

    a. Using Github Desktop:
        i. Once Github Desktop is opened, on the top left corner of the software, click the Current Repository button to bring up the dropdown menu.
        ii. At the top of the dropdown menu, click the “Add” button.
        iii. A menu will appear, click Clone Repository.
            1. **NOTE**: In this menu, the location of where the repo will be stored in the computer can be set at the very bottom of the popup.
        iv. Scroll to find the repo, named “NOBL-Webapp” and click on it.
        v. The Web App should now be stored locally.

    b. Clone using the command line with Git or the Github CLI

2. Open the repo in an IDE of your choice.
    a. **NOTE**: it is recommended to use Visual Studio Code

3. Confirm the the repo looks like the following repo:

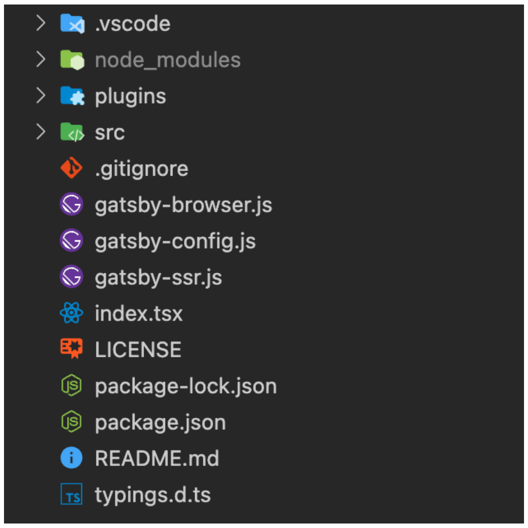


```
> .vscode
> plugins
> src
  .gitignore
  gatsby-browser.js
  gatsby-config.js
  gatsby-ssr.js
  index.tsx
  LICENSE
  package-lock.json
  package.json
  README.md
  typings.d.ts
```

4. Open up a terminal in your IDE.

   a. **Visual Studio Code ONLY**: On the toolbar at the top, click on Terminal and then new Terminal.

5. Within the terminal, enter "**npm install**" on the command line, and run the command. This will then install all the necessary packages that the repo needs which can be seen being installed in the terminal. At the end, a new folder should have been created named "node_modules." Ensure that the repo now looks like the following picture:



```
> .vscode
> node_modules
> plugins
> src
  .gitignore
  gatsby-browser.js
  gatsby-config.js
  gatsby-ssr.js
  index.tsx
  LICENSE
  package-lock.json
  package.json
  README.md
  typings.d.ts
```

6. In the terminal once again, install the Gatsby CLI by entering:

    a. **npm install -g gatsby-cli**

        i. **NOTE: This might require special privileges by using "sudo" in Unix-based operating systems or running Visual Studio Code as an administrator in the Windows operating system.**

NOTE: At this point, the installation of the web application is complete. **HOWEVER**, it will not work with the NOBL API locally since at its current implementation, the Web Application uses AWS Amplify to work with the API. **TO MAKE IT WORK WITH THE NOBL API THAT STORED IN YOUR COMPUTER LOCALLY,** you must make changes to the data handling functions within the repo.

These can be seen in the components specifically for data visualization and will be limited to the pages folder within the src folder, ensuring everything related to Amplify is removed. Once everything has been converted to work locally, you may use "**gatsby develop**" to launch the web app.

To use Amplify which is currently recommended, continue to the rest of the instructions.

NOTE: The following instructions are **ONLY RELEVANT** if you intend to use AWS Amplify.

7. In the terminal, type the following command:
    a. **npm install -g @aws-amplify/cli**

8. Configure you Amazon AWS IAm User to work with the Amplify CLI. Use the following link for a guide on how to do this:

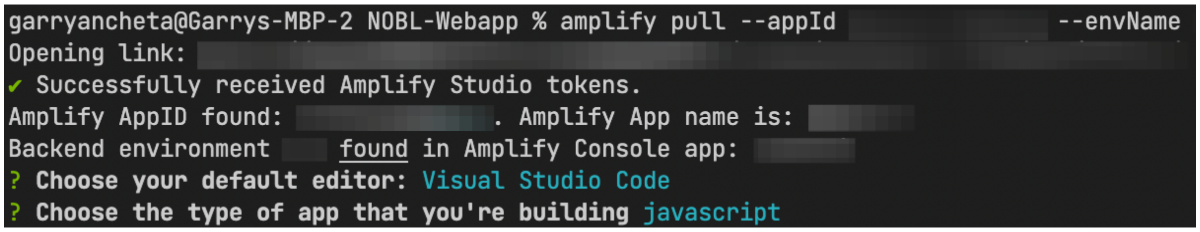
    a. https://docs.amplify.aws/cli/start/install/#option-2-follow-the-instructions

9. In the terminal once again, enter the following command:

    a. amplify pull --appId [REDACTED] --envName [REDACTED]

        i. NOTE: This might open up your web browser to log into AWS Amplify; use your credentials to log in. Once logged in, it should show a success prompt and you can return to the terminal.

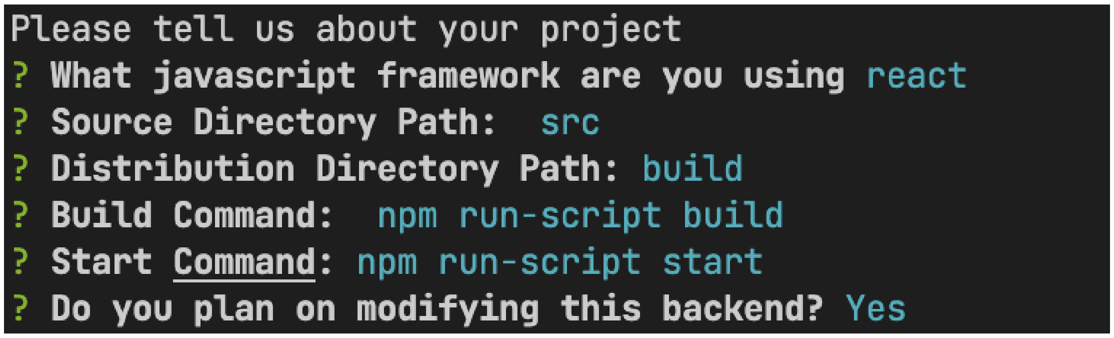
10. The terminal will start to ask questions and confirm what you have requested. At the beginning, this should be what you end up with:



```
garryancheta@Garrys-MBP-2 NOBL-Webapp % amplify pull --appId [redacted] --envName
Opening link: [redacted]
✔ Successfully received Amplify Studio tokens.
Amplify AppID found: [redacted]. Amplify App name is: [redacted]
Backend environment [redacted] found in Amplify Console app: [redacted]
? Choose your default editor: Visual Studio Code
? Choose the type of app that you're building javascript
```

Note: The last two lines are questions that you will be given options to select. The last line must be chosen as javascript.

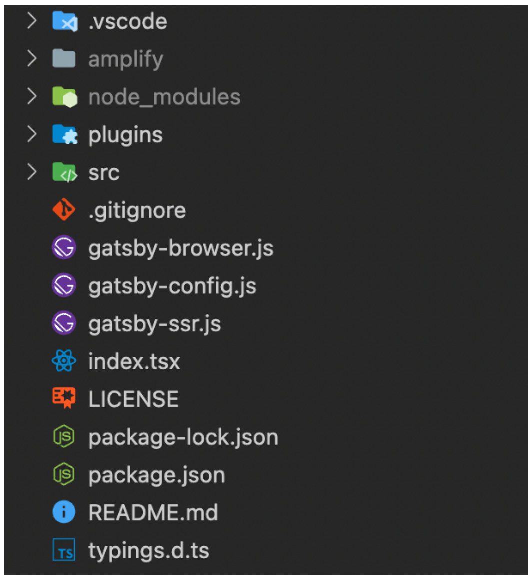
11. The terminal will now ask your more questions that must be answered, the first question in the following picture must have "react" chosen. While the other questions are up to your discretion:



```
Please tell us about your project
? What javascript framework are you using react
? Source Directory Path:  src
? Distribution Directory Path: build
? Build Command:  npm run-script build
? Start Command: npm run-script start
? Do you plan on modifying this backend? Yes
```

12. At this point, the terminal will now start pulling data from AWS. Ensure that the pull was successful on the terminal since it will prompt a success message should the pull be successful.

13. If the AWS Amplify is pulled successfully, the repo should now look like the following picture, notice the new "amplify" folder:

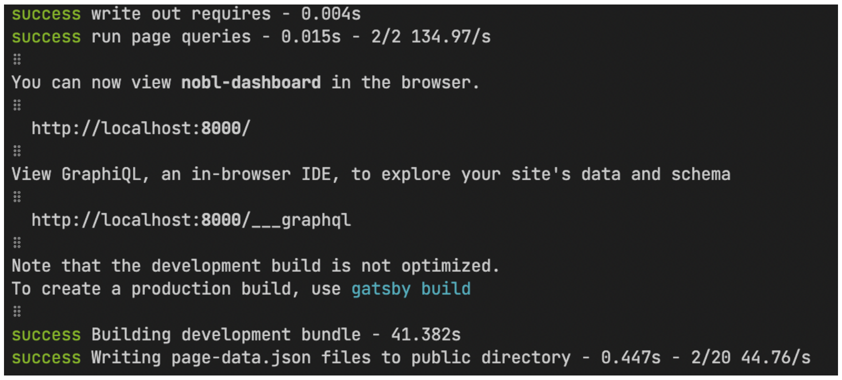


```
> .vscode
> amplify
> node_modules
> plugins
> src
  .gitignore
  gatsby-browser.js
  gatsby-config.js
  gatsby-ssr.js
  index.tsx
  LICENSE
  package-lock.json
  package.json
  README.md
  typings.d.ts
```

14. Now the web application is ready to be run. In the terminal, enter:

    a. **gatsby develop**

15. At the end, should the command be successful, there should be a prompt of where the web application is being hosted. You may now visit the web application through that, as shown below!



```
success write out requires - 0.004s
success run page queries - 0.015s - 2/2 134.97/s
⠀
You can now view nobl-dashboard in the browser.
⠀
  http://localhost:8000/
⠀
View GraphiQL, an in-browser IDE, to explore your site's data and schema
⠀
  http://localhost:8000/___graphql
⠀
Note that the development build is not optimized.
To create a production build, use gatsby build
⠀
success Building development bundle - 41.382s
success Writing page-data.json files to public directory - 0.447s - 2/20 44.76/s
```

Production Cycle

In this section, how the team has allowed for editing any component of the project will be discussed. For the team's production cycle, it consisted of the following phases:

1. Research/Planning
2. Editing
3. Creating a Pull Request
4. Pull Request Review
5. Merging

Research/Planning

Here, the team member would first research the different ways to implement whatever the feature or optimization they are assigned to. This might take a couple of days, but it is to ensure that the team member has a picture of how the goal can be implemented.

Editing

The team member, once research and planning is complete, will now edit the web application or the API depending on what they are working on. The team member should also be, at this point, documenting everything that is being created or edited. Once editing is complete, the team member must ensure that what they have created is tested. The team has a unit testing package that is used to test different parts of the project. Once testing is completed and verified, the team member will now move to creating a pull request.

Creating a Pull Request

A pull request is a request to merge the contributions of a team member to the repo. To do this, the team member must fork the repo and perform a pull request on Github for the original repo. Once a pull request has been made, the team member will notify the Release Manager for review.

Pull Request Review

Once the Release Manager has been notified, the Release Manager will then look over the code and ensure that documentation as well as the code quality is of the standards. If they are not in standards, then the Release Manager will inform the team member who had requested the pull request that there were discrepancies that needed to be corrected.Once the team member has corrected the discrepancies, the Release Manager will be notified and then review the pull request once again.

Merging

Once the Release Manager has approved the pull request, the Release Manager will handle merging the pull request to the main branch of whichever component that the pull request was for. Once the merge is complete, the entire production cycle has been completed.