Software Testing Plan

5 April 2022

Version 1

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

Chapter 1 - Introduction

NOBL Media's Misinformation and Credible News Analysis Tool is meant to help its clients filter out misinformation from their advertising campaigns. However, NOBL Media's clients do not have a way to interface with the Analysis Tool. Team Truthseeker is developing the web application portion of the Analysis Tool by creating the Application Programming Interface (API) and the website application. The API must be tested by making sure its functions are producing the correct output and error free. The API has to pull the right data from the database as well and send information to the website. Next, Team Truthseeker will make sure the app's modules are communicating as expected. The website application portion must be tested to make sure users are comfortable using all the features, as well as being able to easily navigate the page. Applying software testing techniques will ensure Team Truthseeker develops the right web app for NOBL Media.

Software testing is an umbrella term covering many kinds of testing techniques for software. Software testing is necessary for ensuring a program works as intended. Architectural mistakes, design flaws, and security vulnerabilities can be corrected through software testing. Test scopes range from verifying the results of small lines of code and functions to testing if people use the software as expected. Overall, software testing is about giving input and analyzing the results, then making changes if the results of the test do not match expected results.

First, Team Truthseeker will be unit testing the React API. The API's small modular functions will be tested since they make up most of the functionality of the API. Routes, models, and types are our unit tests main focus. The integration testing will consist of making sure our modules communicate well together by simulating correct and incorrect communication and analyzing the results. Usability testing is our final step, where 6 people similar to NOBL Media clients will test out the web application. Testers will try the account and campaign login and authentication feature first. Then Testers will go through the web applications campaign features once logged in. Tester feedback will be considered for changes.

The team is taking a bottom-up approach for software development. Team Truthseekers testing plan starts with unit testing because it is small scale and will help fix any bugs in the basic functionality that will affect the more complicated web application testing. Once Team Truthseeker is confident in the small-scale API functions and results, integration testing can begin since each module's functions are working correctly. This will make sure clients are receiving the correct campaign data. Tester feedback will contribute to reworking the Interface to make it easier for NOBL Media's clients to interfacing with the application. Integration and user testing may lead to more testing in a smaller scale scope to fix any issues affecting the abstracted subject.

Chapter 2 - 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:

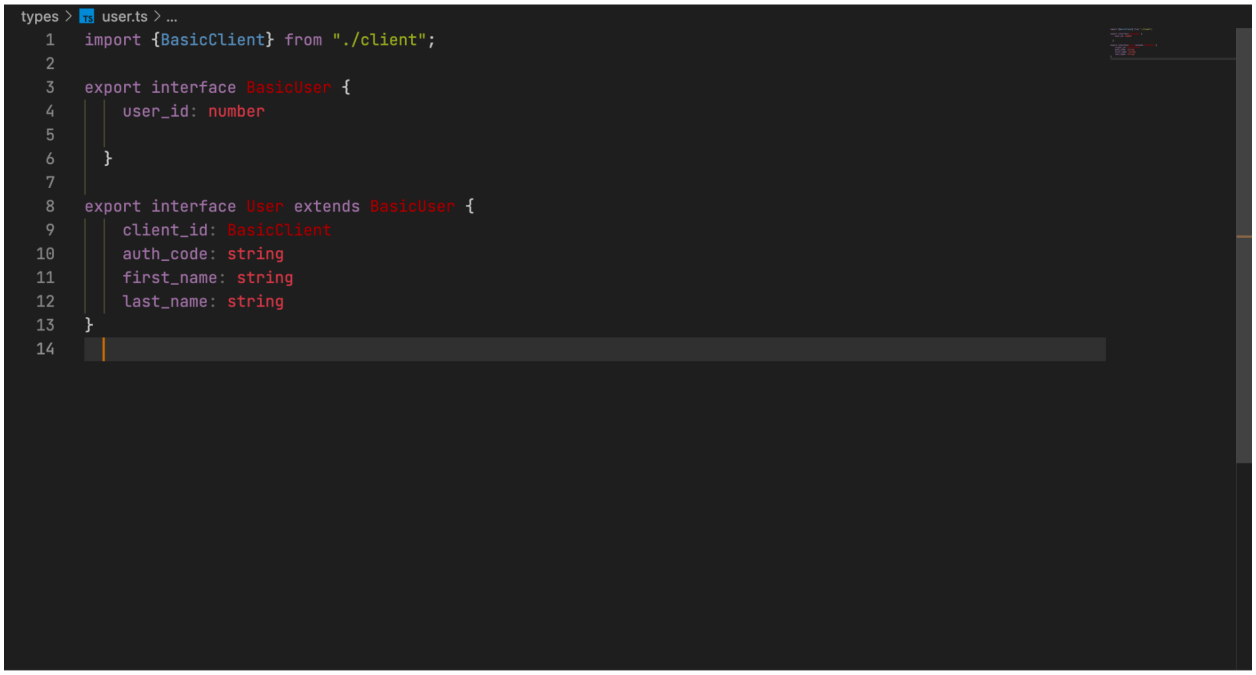
1. Small, modular functions
2. The API's purpose is to output data

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:

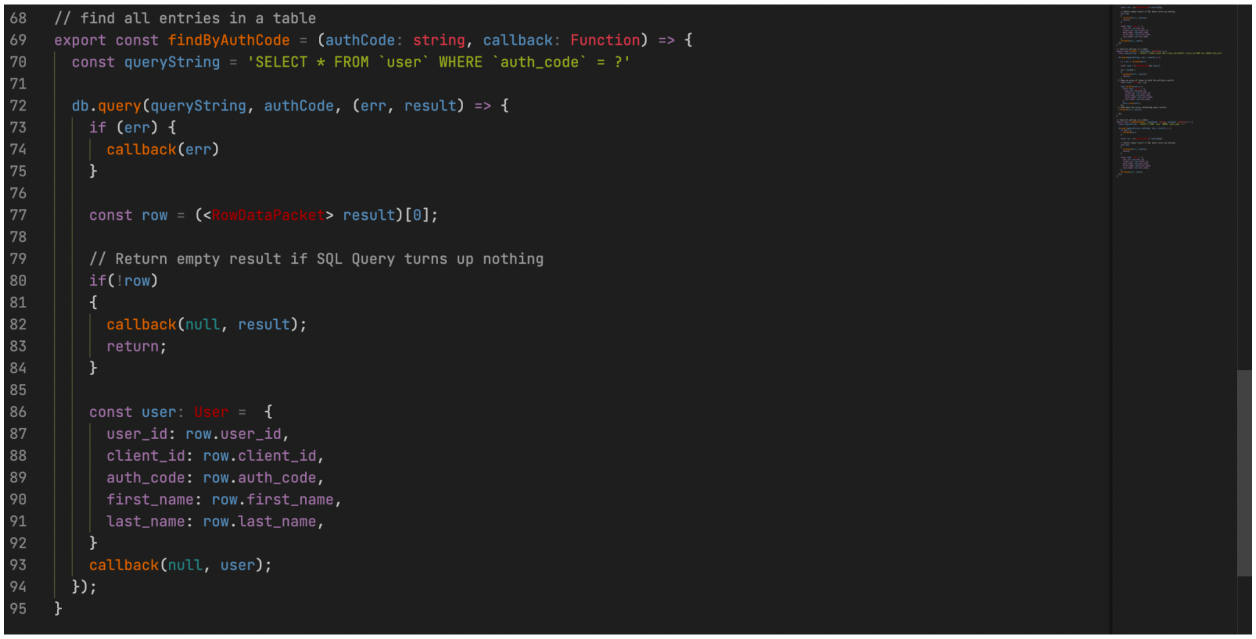
1. Routes
2. Models
3. Types



```
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 2.1 - A User Type

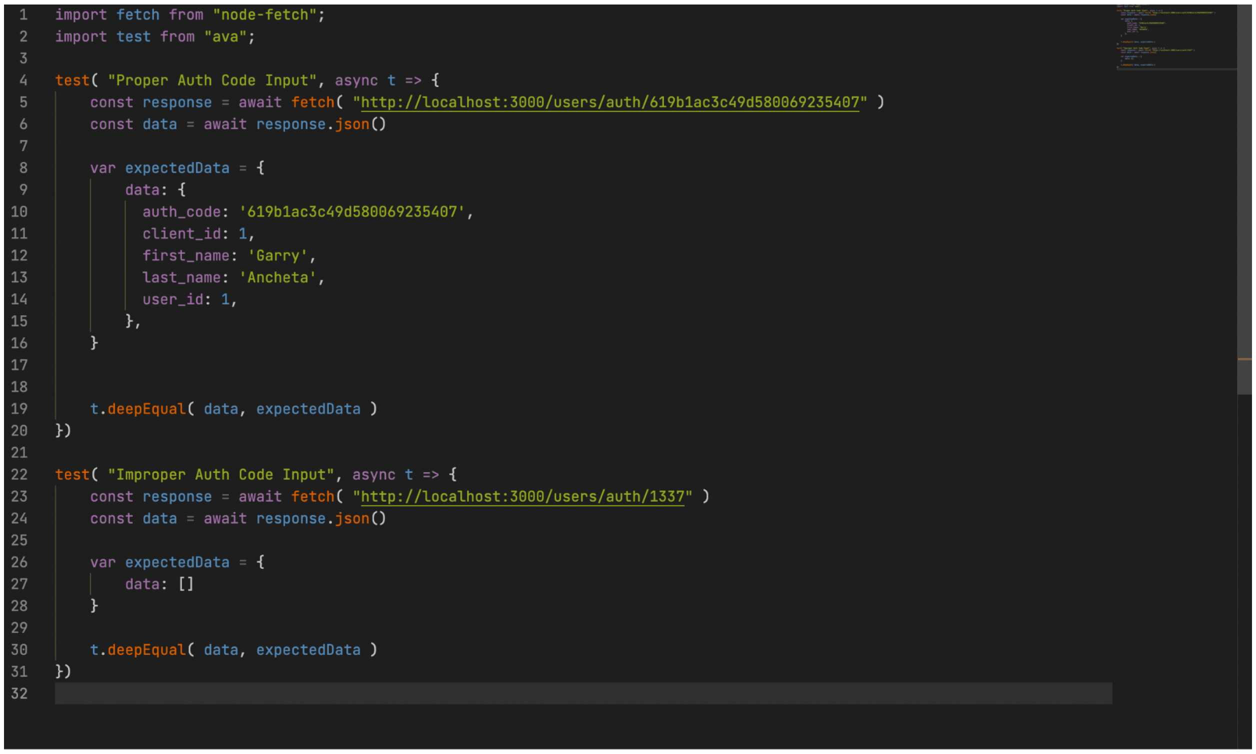
The Routes component allows for the front-end to navigate through the API, allowing it to retrieve specific data that it needs, not just everything the API provides. The Types component allows the API to define what it expects the data to be when it pulls it from the NOBL database. Referring to Figure 2.1, 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 2.1, 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 2.1, 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 2.2 - One Part of the User Model

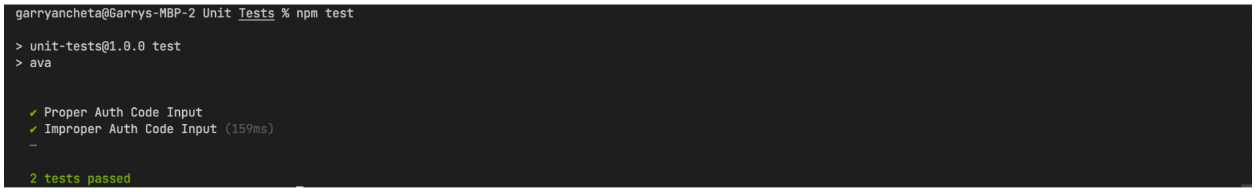
The Models component is where unit testing comes in. The Models component contains the functions that retrieve data from the database. In this case, the unit tests are for when requests from the web application are sent to the API and are waiting for a response. Referring to Figure 2.2, this function 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 2.3 - Snippet of Unit Testing Code

Figure 2.3 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 2.4 - Results of the Unit Test

Figure 2.4 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.

Chapter 3 - 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.

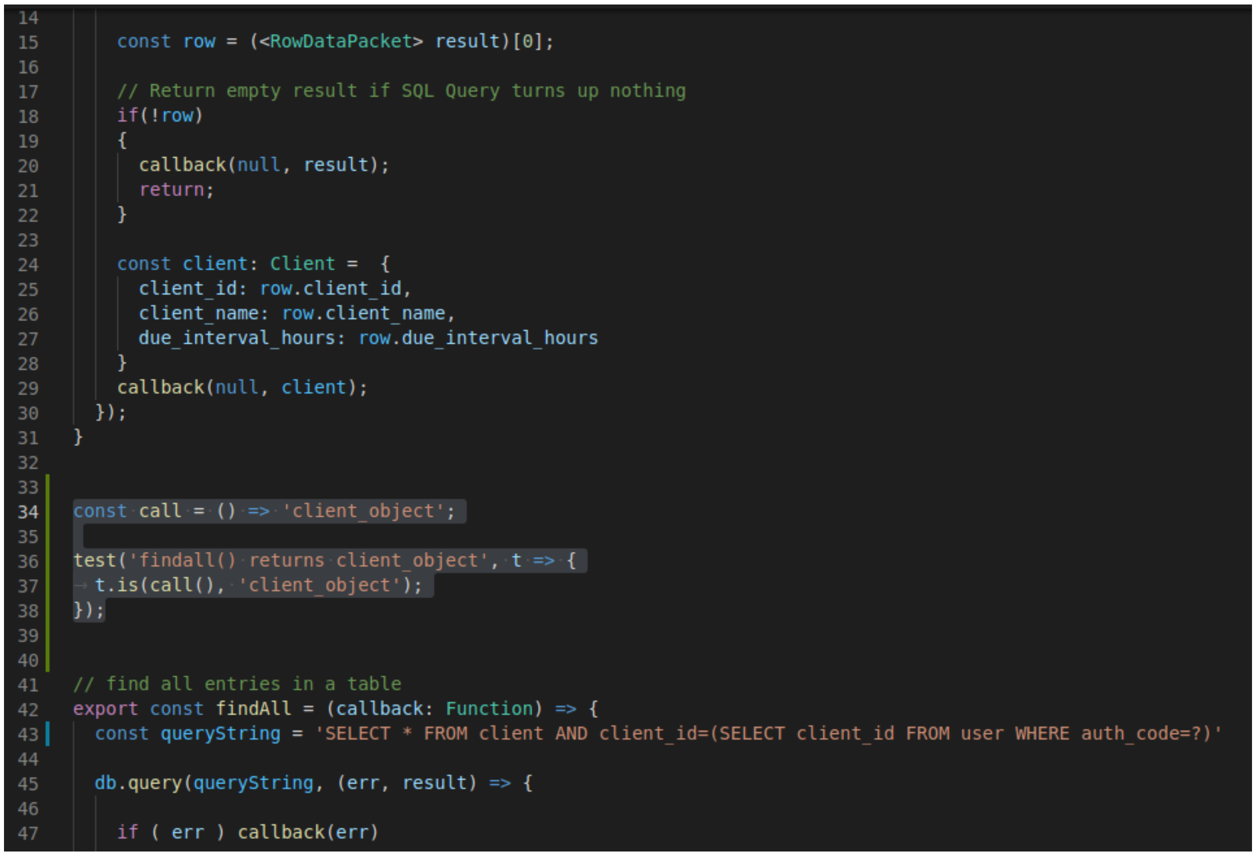
The web application begins by taking client HTTP requests to Auth0's third party authentication server from which is passed an authentication token which logs the user in and displays their information. Here lies the first challenge of testing for proper integration of the third party authentication 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 3.1 - 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.

Chapter 4 - Usability Testing

With the integration testing having been completed, at this point in the project the last test is the usability test which is planned out 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 order to set up a test as such, a selected number of users are chosen to test out various functions throughout the web application. In doing so, the users will describe their experience and answer a series of questions.

To test end-users on the Misinformation and Credible Analysis Tool, at least 6 users need to be selected. Since NOBL Media's clients are not accessible for testing, the criteria of the chosen testers needs 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 need to simulate a similar background. The team plans to send 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 is enough to give adequate feedback, but not too much to dilute the results.

This test will use techniques to ultimately gather qualitative data, where the team will record and analyze user interactions with this product. About 6 users are needed to set up pair testing, which will be used to compare specific user interactions. Each pair test needs to be set up similarly in order to keep the results aligned with one another, so the team plans to write a script to ask the users questions over their interactions and overall experience. This will also guide the usability test to ensure the users interact with specific functionalities that need to be tested and assessed.

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

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

The test will start with each user attempting 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 has attempted the failed log-in, they will select the button labeled as “forgot password” to ensure they are redirected to a new page where they can input their email. Before attempting a log-in with the correct credentials, the user will check to ensure they received an email to reset their password. No user will be prompted to select a new password as this is a functionality on NOBL Media’s database. Once receiving the email, the user will go back to the web application to successfully log-in and in doing so they should be directed to the campaign selector. Here the user can view all campaigns associated with their account and should click on whichever campaign they desire. From here the user is 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 have interacted with the dashboard in these areas, they will select the tab to log out of their account.

The team will have a script to ask the users questions which will guide each user through a similar testing process. While guiding the users through interacting with this product, the team will record their answers to the questions as well as their personal experiences. Once all user testing has been completed and all the results have been recorded, the team plans to compare the results of each pair testing with one another to achieve an overall consensus of user-experience with each functionality.

Chapter 5 - Conclusion

In summary, Team Truthseeker is creating the web application portion of NOBL Media's Misinformation and Credible News Analysis Tool. The web application has gone through the alpha demo stage, so it has most of its functionality. Now, Team Truthseeker has devised a software testing plan to make sure the web application runs as intended. The tests consist of unit, integration, and user testing.

The unit tests that are being implemented are targeted towards the API since it is the "brain" of the software through retrieving data. The API is also ideal for the unit tests due to its small, modular components. Ultimately, unit testing will be used to ensure that integration testing can resume and that proper data retrieval is done. The integration testing ensures proper module communication correctly and error handling which ultimately, validates the web application as a stable product ready for users to try. Auth0 built-in testing to make sure the front-end authentication complies with Team Truthseeker's API. Integration testing also ensures the proper structure of data being pulled by the API from the NOBL Database. Finally, turning HTTP data into a JSON format that the web application user interface can use from the API. User Testing follows module testing since the focus is on the user experience rather than the code. At least 6 testers will test the web application and provide feedback. Login cases and the ease of UI navigation will be the main focus for the testers' feedback which will be used to refactor code to improve the web application's interface.

Team Truthseeker will be working and improving the web application for the next few weeks through testing. The web application has been developed right on schedule. The web application is nearing completion with some stretch goal implementation on the horizon as well. Most of all, the team is working towards getting the project ready for UGrads on April 22nd where the team's client will attend to view the product. All in all, Team Truthseeker is confident in its pacing and progress towards delivering the web application to NOBL Media.