Software Test Plan V.1 - Fall

11/11/2022

Project:

C & I Doctoral Tracking Tool

Project Sponsor:

Gretchen McAllister

Faculty Member:

Michael Leverington

Team Name:

What's Up Doc

Team Members:

Adam Larson (Lead), Brandon Shaffer, and Eddie Lipan

Team Mentors:

Daniel Kramer

Table of Contents

1.0 Introduction 3

2.0 Unit Testing 4

3.0 Integration Testing 12

4.0 Usability Testing 15

5.0 Conclusion 16

1.0 Introduction

Team What's Up Doc is tasked with producing a practical and approachable website application that will allow graduate students within the NAU Curriculum and Instruction (C&I) Doctoral Program to track their own progress and allow administrators to analyze how students are doing within the program. Graduate students will be able to access a visually appealing dashboard that presents them with the different phases required to complete the program. Students will be able to tab between phases and look at individual milestones within each phase. Within these individual milestones, students can download blank copies of the documents needed to complete the milestone, or students can upload completed documents. Once a completed document has been uploaded, the dashboard will visually update the "milestone task bar" at the left side of the screen to indicate that the milestone has been completed. Once all milestones have been completed within their respective phase, the appropriate phase tab within the "phase progress bar", at the top of the screen, will change colors to reflect this completion.

This website application is also intended to make the administrators lives easier as well. Initially, administrators were tasked with receiving and tracking all relative deliverables for the students within the program. This led to a large amount of overhead as students had to inquire about deliverable completions via email interactions and administrators had to actively track and find deliverables in their local file directory, which is time consuming. Our application will place the "data entry" responsibilities on the graduate students themselves which frees up time for the administrators to focus on program content or program analysis.

With these motivations in mind, Team What's Up Doc will be performing extensive software testing. Software testing is carried out in order to assure that our product is doing what it is supposed to be doing. For months now we have been meticulously designing and integrating different ideas and systems into a single cohesive website application, and now we have to prove that our work does what is intended. Not only is software testing an assurance of sorts, but it also protects us from bugs, reduces future development costs and improves application performance.

With that in mind, this document will now segue into the specific testing being performed for our application and how it is being done. Utilizing tools such as Chrome DevTools, Postman, and Spring frameworks, the developers are able to carry out a plethora of tests. Chrome DevTools helps with Unit Testing, Integration Testing and Usability Testing, while Postman and Spring Boot help a lot with Integration Testing. The particular units being tested will focus on user security, user uploads, and making sure

that the correct sensitive student information is being shown to the right people. This will require access tokens and a good stream of communication between the web application and the database server. Considering our application is related to data entry, data tracking and data retrieval, most of our testing will relate to making connections to our database.

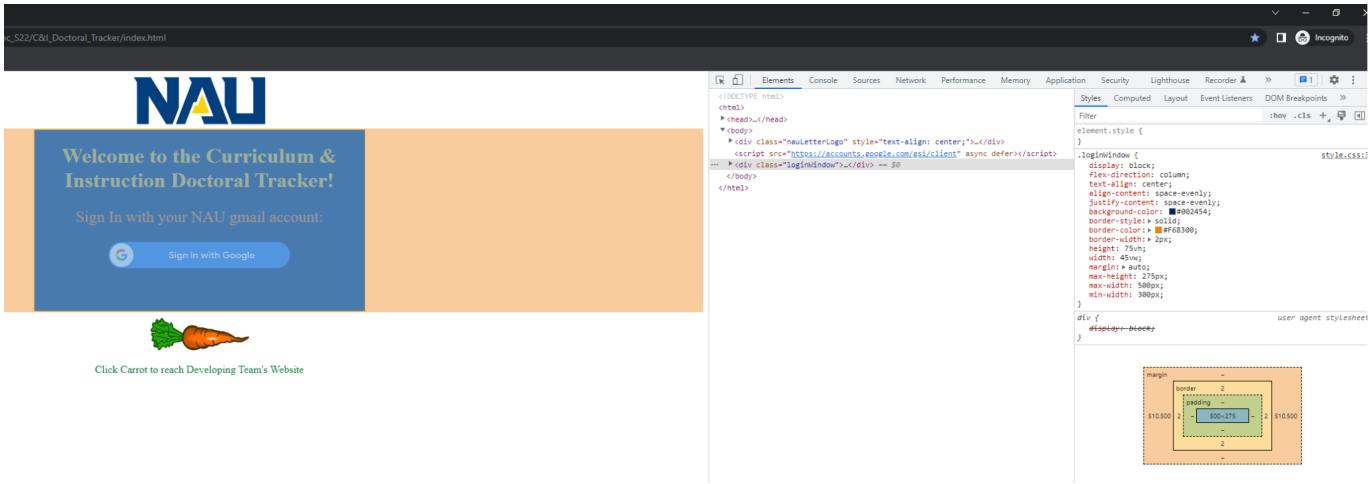
2.0 Unit Testing

In software, a unit is the smallest testable component of an application. Therefore, completing unit testing entails making sure that particular methods, objects, and packages used in the code are working as intended. Unit testing is generally performed by the developer during the development process and can be done manually or automatically. The units being tested for this application are as follows: sign in with Google button, accurate redirect to appropriate home page after sign in, accurate exchange of JWT, accurate upload of unique student files, accurate download of unique student files, accurate updating of milestone/phase colors on webpage (completed phases/milestones are gold), accurate requests made to database in order to populate admin tables, CRUD capabilities for student table, and a proper search bar that allows the admin to view a students unique home page embedded within the admin home page.

2.1 Website

Most of the team's unit testing for the front end has been done manually. This is solely due to the fact that the developers are coding these web pages using HTML, JavaScript and CSS and rely heavily on manipulating the pages through minor tweaks in the code. After a minor change in one of the local source files is carried out, we update the main server with the updated source file and refresh the web page to view our code change and its effects in real time. This leads to a general back and forth between writing code and pushing the changes to assure that the webpage is reflecting the changes you intend to make. Naturally, this coding process is very friendly to unit testing and allows the developer to assure that the code they have written is functioning properly before moving on to another part of the web page.

In particular, using tools such as Chrome DevTools allows the developer to view their web page in real time and isolate particular elements for testing purposes. In graphic 2.1.1, you can see the landing page to our application and the user is highlighting a line of code from the index.html source code in the Chrome DevTools "Elements" window:



Graphic 2.1.1: Chrome DevTools in action

The user is hovering over a div in the html code that encompasses the entirety of the “login window”, hence why Chrome DevTools is highlighting the respective login window on the left side of the graphic. This is useful for unit testing because it allows the developer to map what code is affecting a particular part of the web page. For example, if the developer had intended for the carrot graphic, seen above in Graphic 2.1.1, to be within the login window instead of outside the login window, this tool explicitly lets the developer know that the code for their carrot graphic is not in the intended place. Chrome DevTools allows the developer to view other windows that relate to the console, network, performance, security, and memory of the web page, all of which are invaluable for unit testing. We will highlight a few of these DevTools tabs in the sections that follow. With Chrome DevTools in mind, we will shift to each individual web page being used on the front end and articulate the relevant unit testing being performed for that page.

2.1.1 Landing Page - index.html

With regards to the web application’s landing page, there are two noticeable features that involve unit testing. The first feature involves correctly incorporating Google Identity Service’s code so that the recognizable “Sign-In with Google” button appears correctly within the login window. Graphic 2.1.1.1 illustrates Google’s code in the index.html file, with sensitive information temporarily removed:

```
<div id="g_id_onload"
    data-client_id="INSERT DATA CLIENT ID"
    data-auto_select="false"
    data-auto_prompt="false"
    data-context="signin"
    data-ux_mode="popup"
    data-login_uri="INSERT REDIRECT URI"
    data-itp_support="true">
</div>

<div class="g_id_signin"
    data-type="standard"
    data-shape="pill"
    data-theme="filled_blue"
    data-text="signin_with"
    data-size="large"
    data-locale="en-US"
    data-logo_alignment="left"
    data-width="275">
</div>
```

Graphic 2.1.1.1: Google's Sign in button code in use

With a proper amount of research into Google's extensive documentation and some manual unit testing, the developers were able to incorporate Google's sign in button seamlessly into their web page. The developers were also able to customize the button's size, shape, color as well as where the button would redirect to upon sign in.

Unit testing was conducted specifically on the *data-login_uri* variable to ensure that when a user logs in using their NAU gmail, the home page they get sent to is in fact their home page and not another student's home page. Considering this is a College of Education web application, the security and privacy of the students and administrators is of utmost importance and we cannot allow students to be redirected to another student's home page. The unit testing for this feature entails receiving a JSON Web Token from Google after the user has successfully logged in with an NAU associated gmail account, properly decoding the JWT, stripping the user's NAU userID from the decoded JSON body and passing it in the appropriate spot within the *data-login_uri* string.

The second feature that needed extensive unit testing relates to the JSON Web Token (JWT) mentioned above. The developers need to ensure that the encoded JWT response received from Google is properly decoded. Without properly decoding the JWT, the developers will have a payload they cannot manipulate. Once the JWT payload has been decoded, the relevant access tokens and user information need to be

exchanged with the back end database in order to ensure that the user ends up at their own respective home page and that when they make a change on their student home page, it is properly reflected on the back end. This feature is crucial for the overall security of the website application and must be completed with care.

2.1.2 Student Home Page - home.html

The student home page is hosted on the home.html file. The relevant unit testing for this involved the various database interactions and the interactivity of the website itself. The page needed to dynamically change color based on the database, and change its layout based on user interactions. The layout of the webpage is a series of nested drop downs that, when clicked, needs to show the relevant submenu. The website, on loading, needs to contact the database and retrieve a list of the logged in user's complete list of uploaded files. For testing purposes, the retrieved list is being displayed on the console log so that any bugs can be troubleshot. Should an element not react to the presence of its corresponding file, we can check the retrieved list visually to ensure that the request was correctly made. The displayed list also aids the testing of the upload functionality of the website.

The website will need to be able to upload a file from the local computer to the remote database, and store it under the correct name and user. Testing this functionality required uploading random files and checking the list of uploaded files for the corresponding file. When the download function was implemented, that too was used for testing by downloading the uploaded file.

Downloading files from the database via the website was the last major functionality that required major testing. Testing the download function consisted of uploading files under various categories and then downloading them, and ensuring that they are the correct file.

2.1.3 Admin Home Page - admin.html

The administrator's home page resides within the *admin.html* file and the unit testing relevant for this file involved the different tables being used on this page and being able to tab between the three features available to the admin. The administrators have access to three buttons which lead to two tables and a search bar. The first button should bring up the "Add/Remove Students" table. This table allows the admin to add/edit/remove students to and from the graduate program. This is a vital feature of the page as the admin has total control over which students are in the program and as such, they are the gateway for the student to be able to use this web application. Administrators must be able to add/edit/remove students on the front end and have it visually update the table in real time. Administrators need to be able to accurately sort students by any column they want and they should be able to search the entire table by

keyword. If the admin clicks the second tab at the top of their screen, labeled “Phase Review”, it will bring up the second relevant table. This table is linked to the first table and as students are added to the first table, they must show up in the second table as well. Lastly, the third tab labeled “Student Progress Viewer” should allow the admin to search the program for a student, by their userID (explicitly) and it will bring up an embedded view of that student's unique home page with their completed milestones. This page is very graphic intensive and when the user does an action it needs to be accurately reflected in real time on the appropriate table.

2.2 Server

Using Spring Boot as our REST API framework, a significant amount of support is available for testing. It provides us with the ability to run unit tests using either JUnit4 or JUnit 5 and using a mock servlet environment. These tests require little set up as the framework allows some simple annotations to be placed for appropriate execution.

Unit testing in particular is very easy in that controller code is the only part needing testing. All methods through the program are run through the controller when an endpoint is accessed. Simulation of requests to these endpoints as well as pseudo-data allows each to be tested for normal and fringe inputs. An example of this can be seen in the following:

```
@RestController
@RequestMapping("/api")
public class EmployeeRestController {

    @Autowired
    private EmployeeService employeeService;

    @GetMapping("/employees")
    public List<Employee> getAllEmployees() {
        return employeeService.getAllEmployees();
    }
}
```

Graphic 2.2.1: A controller example with basic functionality

```
@RunWith(SpringRunner.class)
@WebMvcTest(EmployeeRestController.class)
public class EmployeeRestControllerIntegrationTest {

    @Autowired
    private MockMvc mvc;

    @MockBean
    private EmployeeService service;

    // write test cases here
}
```

Graphic 2.2.2: Class set-up for test cases
(Ignore the integration test class name - the tutorial being referenced combined them and named the class for the initial testing)

```
@Test
public void givenEmployees_whenGetEmployees_thenReturnJsonArray()
  throws Exception {

    Employee alex = new Employee("alex");

    List<Employee> allEmployees = Arrays.asList(alex);

    given(service.getAllEmployees()).willReturn(allEmployees);

    mvc.perform(get("/api/employees")
      .contentType(MediaType.APPLICATION_JSON))
      .andExpect(status().isOk())
      .andExpect(jsonPath("$", hasSize(1)))
      .andExpect(jsonPath("$[0].name", is(alex.getName())));
}
```

Graphic 2.2.3: Test case for the above class

Unit tests set up in such a way allow us to take advantage of the framework's testing suite and its use of annotations to create requests and act on mock data. The @Autowired annotation creates an object that can be acted on to make http requests and match details of responses to pass or fail. @MockBean on the other hand provides a bootstrapped instance of the necessary classes for a test case.

In terms of our products end points, several tests will need to be performed on each to provide adequate confidence on proper functionality. For normal requests retrieving JSON information, the tests would include standard operation, incorrect formatting and types, and using alternative request types. Authentication access errors would also be tested for in this area, though due to time constraints and challenges, these will not be present in actual testing. Posting, deleting, and updating data tests will function similarly but an additional check for filtering things like SQL injection.

The goal is to pass all tests run, and modify as necessary. Some assumptions of additional tests are present, but not accounted for within this document and may arise during testing sequences. Additionally, the Spring framework offers a number of auto-configured tests flagged with specific annotations. While no plans to use them currently exist, there is consideration to include these if necessary.

2.3 Database

Unit testing of the database is both challenging and lightweight. It is challenging in the fact that while MySQL provides some information on implementation within their documentation, it doesn't seem to provide enough to quickly set up a test suite like the server. On the other hand, it is far more lightweight as testing should go quickly using various sequel statements to ensure proper input, output, and sorting occurs.

As of writing, there appears to be two quick ways to implement this. First would be a simple shell script that performs sql commands and outputs everything from the test file. This would be very quick to set up, however passing and failing would be manual and it would likely be just as quick to manually make the sql calls and inspect each for proper functionality. The second and probably more reasonable option is creating a view from a SQL query then creating a test-query within it. This view can contain multiple tests, which can output either individual results or create a separate database containing the results. This option sounds significantly better than manually, however, this was the only quick implementation found - many suggested writing their own libraries to automate unit testing regularly. Unfamiliarity with both view queries and scripts within MySQL makes this more difficult to understand if and how to implement it.

The testing plan will consist of this latter method first, with the fall back of manual testing and documentation of the process and outputs. Examples of the implementation are as follows:

```
1  create or replace view
2  my_query
3  as
4  select personid
5  ,      to_char(login_time,'HH24:MI') block_start_login_time
6  from logins
7  match_recognize (
8    partition by personid -- regard every person independently
9    order by login_time
10   measures
11          match_number()     mn,
12          block_start.login_time    as login_time
13   one row per match -- every block is represented by a single record
14   pattern (block_start same_block*) -- collect a log in record and all subsequent rows in
15   define
16        same_block as (login_time <= block_start.login_time + interval '2' hour)  -- a logir
17 )
```

Graphic 2.3.1: Creating the view (very low resolution only - sorry!)

```
1  with result_to_verify as
2  ( select block_start_login_time  login_time
3    from   my_query
4    where  personid = 1
5  )
6  , expected_result as
7  ( select '00:00' from dual
8    union all
9    select '02:39' from dual
10   union all
11   select '04:59' from dual
12 )
13 (select *
14 from result_to_verify
15 minus
16 select *
17 from expected_result
18 ) -- superfluous results
19 union all
20 (
21 select *
22 from expected_result
23 minus
24 select *
25 from result_to_verify
26 ) -- missing results
```

Results | Explain | Describe | Saved SQL | History

no data found

Graphic 2.3.2: Sample code to include within the view

As stated prior, implementing this testing depends heavily on quick understanding of unfamiliar sequel tools. It does seem plausible however, and results would provide “no data found” on a successful test. If this was unable to be

implemented successfully, results of manual tests would be required to prove successful operation. Results would primarily need to include the select, insert, update, and delete functions.

3.0 Integration Testing

Proper integration testing is a vital component to the success of this web application. Without proper integration testing, users would never be able to make it past the landing page of our web app. Therefore, we must ensure that our web app is in contact with our database server.

This web application requires the user to sign in using a verified NAU gmail, and once that has been carried out, the application must then recognize whether the user is an admin or a student. Student's must be redirected to their unique student home page and admins must be redirected to the admin home page. Any mistakes here would lead to grave security concerns. Once past the landing page, the web application relies on the back end database to populate tables for the admins or update the dashboard for the student with accurate and current data. Without Integration testing and a successful discourse between the front end and back end, our web application would be nothing but static pages.

3.1 Website - Server

The integration testing carried out on the front end was aided by Chrome DevTools and Postman. Initially, Postman was used to ping our database and receive initial contact. Once a proper "GET" request could be made from Postman to our server, the developers began making requests to the database in the appropriate JavaScript files within the web application.

The index.html file, that renders the initial landing page, has one connection to the database. The index.html file, upon successful login by a student or administrator, must be able to receive a JWT payload from Google servers and decode the JWT payload. Then, the appropriate user information, such as first name, last name, userID, email and access tokens must be sent to the back end for verification. Once the access tokens and user information have been verified, the user's userID and any relevant access tokens are sent back to the front end so that the user can be properly redirected to their unique home page or the admin home page.

The home.html file, that renders the student's home page, must be able to accurately and securely lead the student to their unique home page from the landing page. The student should then be able to upload completed deliverables to their milestone dashboard, which must also send the file to the database to be saved. Any

time the student loads their unique home page, the web page must be in contact with the database in order to accurately recognize which milestones have been completed and, as a result, accurately change the colors of the progress bar/task bar to reflect what has been completed.

Finally, we reach the admin's home page which is contained within the admin.html file and admin.js file. Again, the admin must be properly redirected to this page from the login page, and this must be done securely. Students should not be able to access the admin home page under any circumstances. Once an admin has been properly redirected to this web page, they should have access to two tables. Both tables are constantly receiving data from the database server via “GET” requests. Since the developers implemented CRUD functionality on the back end, they decided to implement their own CRUD buttons for each table using “POST” and “PUT” requests where necessary. As CRUD functions are carried out, the action needs to be reflected on the back end, which in turn alters the table on the front end as well. Our front end is basically a Graphical User Interface (GUI) that has to look and feel like the user is making changes on the front end and know that their changes are being saved on the back end as well.

3.2 Server - Database

Integration testing is made easy through Spring’s test suite, very much like its unit testing. It’s nearly identical in set up and function, though it includes the connection to the database through either the CRUD or JPA repositories used by the program. Set up does require modifications to a testing application.yml (or application.properties) file as the current test plan includes using in-memory H2 persistence storage. To add this to the suite the following must be included within the properties file:

```
spring.datasource.url = jdbc:h2:mem:test
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.H2Dialect
```

Graphic 3.2.1: Application.yml (or .properties shown) modification for H2 persistence

As mentioned above, set up for integration tests include some of the very same elements as unit testing. Where these vary is within communication with the database and what is being used as the test element. Instead of sending a mock element into the test to have it acted on and compared, we will be sending an object of normal data. Examples of the annotated class and test case are as follows:

```
@RunWith(SpringRunner.class)
@SpringBootTest(
  SpringBootTest.WebEnvironment.MOCK,
  classes = Application.class)
@AutoConfigureMockMvc
@TestPropertySource(
  locations = "classpath:application-integrationtest.properties")
public class EmployeeRestControllerIntegrationTest {

    @Autowired
    private MockMvc mvc;

    @Autowired
    private EmployeeRepository repository;

    // write test cases here
}
```

Graphic 3.2.2: Example annotated class
(named the same as the unit class - more annotations here)

```
@Test
public void givenEmployees_whenGetEmployees_thenStatus200()
  throws Exception {

    createTestEmployee("bob");

    mvc.perform(get("/api/employees")
      .contentType(MediaType.APPLICATION_JSON))
      .andExpect(status().isOk())
      .andExpect(content()
      .contentTypeCompatibleWith(MediaType.APPLICATION_JSON))
      .andExpect(jsonPath("$[0].name", is("bob")));
}
```

Graphic 3.2.3: Example test case

Using the additional annotations to include the bootstrapped program with H2 persistence, and with the simple test case inclusion of an item to be saved, allows integration to be easily tested. It is nearly identical to unit testing within Spring's suite of testing, but with few key differences.

Test cases for implementation within these systems is primarily based on proper data being moved from the CRUD/JPA repositories to and from the database. These will follow suit of the unit tests, with a significant sign of failure if one passes and the other does not.

4.0 Usability Testing

Usability testing is the process that intends to ensure a product's viability with the end user. Since our potential users cover a wide variety of ages and technical skills, we will need to ensure that the interface is as accessible as possible. Due to the fact that accessibility is more of a subjective metric, we have and will rely on focus group testing with users.

4.1 Website

To test the usability of the website we have been meeting with our client weekly. At these meetings we familiarized her with the recent updates to the system's functions and layout. In the upcoming three weeks, we plan on setting up a focus group with the client's students over zoom. With this focus group we will gather qualitative data on the user interface. We will start the session by going through the basic functions of the student-side interface. After going through the interface with the students, we will proceed to answer any questions from the group. Then we will ask for their opinions of the interface. During the focus group, one of us will be taking down notes of the questions the group asks, and their feedback. After the focus group, we will look through the notes and see what improvements we can make, based on the feedback and questions, within the time we have left.

For the client meetings we have been doing something similar in that we walk through the interface and let them ask questions before asking for specific feedback. The key difference here is that we show them the administrators interface as well. We will continue to meet with the client to fine tune the interfaces' layouts.

5.0 Conclusion

Team What's Up Doc was tasked with creating an approachable application for NAU's C&I Doctoral Program, for the purpose of tracking students' progress with their doctorate. The application required a remote database for the storage of students' files, as well as a front-end website for users to interact with the database. The website will need to upload and download from the database, as well as provide an interactive interface to display progress for users. The website would also need a separate interface for faculty users to check the progress of various students and search the database for various statistical metrics via spreadsheets. To ensure that our product meets the requirements as set by our client, we performed various tests on each element of the product. The database was tested for its file storage and responsiveness, while its connections were tested with the Spring test suite. The front end's tests, consisting of focus groups as well as functionality tests, will help ensure the product will be completed with error-free functionality.