Software Test Plan

WearWare Study Manager

Sponsored by Dr. Kyle N. Winfree and Dr. Eck Doerry
Mentored by Han Peng

Members:
Anton Freeman, Halcyon de la Rosa, Karina Anaya, Noah Olono

Version 1.0
4/6/2022

Table of Contents

1. Introduction

Health research is a very important field, as insights gained from studies into health issues like heart disease can help save many lives. Currently, health research often uses expensive clinical tools like heart monitors and fitness trackers to collect information from study participants, but these tools are often very expensive and difficult to use, which can result in researchers not getting as much data as they would like. We are working on an alternative method of health data collection through the use of Fitbits, which are inexpensive wrist-mounted consumer devices which can gather the same data that the aforementioned clinical tools get but at a much lower cost while also being more convenient to use. Specifically, we are making a secure web application that will allow researchers to collect Fitbit data from study participants, manage their studies and monitor participant data, and then finally export participant data for use in their research.

In order to make sure that our project works correctly and is secure, we need to do some software testing. Software testing is, as its name implies, a process in which we will take the different parts of our project and give them several “tests” to ensure that they work as intended without any bugs or problems that could cause issues with our project’s final product. These tests can involve things like using debugging tools, sending randomized input into our different software functions to see what causes them to break, and making sure that our end users are able to use the software as intended without any confusion or problems. This is important because while we are developing software it is very easy to accidentally let certain issues slide past our notice, which could potentially range anywhere from a mild annoyance to making our project completely unusable. Regardless of the impact, we want to make sure that our project does not have any issues that could potentially upset our client and thus we need to test our software to ensure that it works properly.

Our current testing plan involves three main types of testing: unit testing, integration testing, and usability testing. Unit testing is a type of testing in which we test certain “units” of code to ensure that they work properly, such as individual functions or components of our web app. We will be conducting unit tests on all the major parts of our application. Once unit testing is finished and we can ensure all of the different parts of our project work on their own, integration testing makes sure that all of the different parts of the application interact with each other properly without any unexpected or unwanted issues resulting from their interactions. This will also be conducted on all the parts of our system that interact with each other, especially those that interact with the main server. Finally, usability testing is not for our code but for the product itself, making sure that our product can be used by our end users without any difficulties or confusion. This will be a high priority for testing, as our client has emphasized the importance of our project being usable for users who are not necessarily technically savvy. With all of these parts together, we will be able to make a solid product for our client.

The reasons for our current testing plan are as follows:

- Individual unit tests are important, but only focusing on smaller units could result in missing the issues that arise when the entire project as a whole comes together.
- Integration testing thus is a high priority, but these are more useful to discovering where a problem lies in a certain interaction rather than the specifics of what is actually causing the problem. As a result, we will be using integration testing to find problems with component interactions, and then unit testing on the individual components to diagnose the specific issue.
- As a result, individual unit tests will be used more than integration tests due to the amount of individual components that exist within the system, although integration tests will also be important.
- Meanwhile, usability testing is a very high priority for us. Our client has specified that our users will not be tech-savvy and thus will need the project to be as usable as possible as to not be confusing to them.

Now that we have laid out the types of tests we will be doing and our reasons for prioritizing different types of testing, we will move on to the actual examples of what tests we will be using.

2. Unit Testing

In this section we will outline our unit testing plan for key methods of our systems components. Unit testing is a type of testing which emphasizes testing the functionality of an individual method or small section of code, referred to as “units”. This testing at the simplest level is taking a pre-formed input, passing it to the target method or section, and then comparing the result returned to an expected result. Our main tool for unit testing will be Jest, which is a tool designed for unit testing Javascript programs. Since our entire project is written in Javascript, we have determined that this is the best tool for the job. We will also be aiming for as high of a code coverage as possible with these tests, which will ideally be 100%. Below is a comprehensive list of the different parts we will be testing in our project.

2.1 Frontend Web App

For our frontend web application, our units of code will be every component that we have. This essentially means that each web page within our project will be tested individually so that we can ensure that they all work correctly on their own before hooking them up to the rest of our project. Since most of our input for the frontend is retrieved from requests to our backend server and these unit tests will only focus on the individual units of our frontend, we will be making heavy use of Jest’s ability to mock other functions. This will allow us to send pre-formed inputs to our frontend web application instead of having to reach out to the server and get inputs from there. In cases where the users give inputs, we will be using Jest’s functionality to put input into form boxes and then see what happens.

2.1.1 AddStudy

AddStudy is a component that allows users to add studies via a form. This function will be tested by using Jest to enter inputs into the form and then passing the result into a mock function that will view the form output and ensure that it exactly matches the input.

- **Equivalence partitions:** Valid dates (end date after start date), invalid dates (end date equal to or before start date). String validation will be handled by request sanitizers, not the form.
- **Boundary values**: Equal start date and end date, start date before end date, start date after end date.
- **Selected inputs:**
    - Load page
    - Valid form inputs:

        - {title: “Title”, consent_form: “This is a consent form”, start_date: 4/13/2021, end_date: 6/12/2021}
    - Invalid form inputs:
        - {title: “Title”, consent_form: “This is a consent form”, start_date: 4/13/2021, end_date: 4/13/2021}
        - {title: “Title”, consent_form: “This is a consent form”, start_date: 6/12/2021, end_date: 4/13/2021}
    - Click confirm button to send form information

2.1.2 ConsentForm

ConsentForm is a component that shows a researcher-specified consent form that either allows users to accept or decline it, with accepting bringing them to the SelfEnroll page and declining preventing them from continuing. To test this, we will use Jest to mock a consent form from the backend and then have it click on the allowed buttons.

- **Equivalence partitions:** There are no equivalence partitions for this component because all input is done via pre-established buttons.
- **Boundary values**: User accepts, user declines
- **Selected inputs:**
    - Load page
    - Click the accept button
    - Click the decline button

2.1.3 EditStudy

This component acts the same as AddStudy but modifies studies instead of adding them. As a result, all testing done will be the same as that in Section 2.1.1, with the exception of two added buttons: one to cancel, and one to.

- **Equivalence partitions:** Valid dates (end date after start date), invalid dates (end date equal to or before start date). String validation will be handled by request sanitizers, not the form.
- **Boundary values**: Equal start date and end date, start date before end date, start date after end date.
- **Selected inputs:**
    - Load page
    - Valid form inputs: {title: “Title”, consent_form: “This is a consent form”, start_date: 4/13/2021, end_date: 6/12/2021}
    - Invalid form inputs: {title: “Title”, consent_form: “This is a consent form”, start_date: 4/13/2021, end_date: 4/13/2021}, {title: “Title”, consent_form: “This is a consent form”, start_date: 6/12/2021, end_date: 4/13/2021}

    - Send form
    - Click cancel button
    - Click delete button

2.1.4 EnrollParticipant

This component allows users to enter a participant's information and enroll them in a study. Participants can have names, birth dates, heights, and weights entered in manually - we must test to ensure that these are possible and that the form is sent correctly.

- **Equivalence partitions:** Valid birth dates, invalid birth dates, valid heights, invalid heights, valid weights, invalid weights.
- **Boundary values**: start date after current date, start date before current date, start date equal to current date, negative weight, positive weight, negative height, invalid height
- **Selected inputs:**
    - Load page
    - Valid inputs:
        - { name: "Dave", participant_id: 001, birth_date: 4/13/2000, height: 21, weight: 22}
    - Invalid inputs:
        - { name: "Dave", participant_id: 001, birth_date: current_date, height: 21, weight: 22}
        - { name: "Dave", participant_id: 001, birth_date: 4/13/2000, height: -21, weight: 22}
        - { name: "Dave", participant_id: 001, birth_date: 4/13/2000, height: 21, weight: -22}
        - { name: "Dave", participant_id: 001, birth_date: 4/13/2222, height: 21, weight: 22}
    - Send form

2.1.5 SelfEnroll

The SelfEnroll component works in the exact same way as the EnrollParticipant component, but automatically generates a participant id. As a result, all testing will be the same as in section 2.1.4, but without a participant id.

- **Equivalence partitions:** Valid birth dates, invalid birth dates, valid heights, invalid heights, valid weights, invalid weights.
- **Boundary values**: start date after current date, start date before current date, start date equal to current date, negative weight, positive weight, negative height, invalid height
- **Selected inputs:**

    - Load page
    - Valid form inputs:
        - { name: “Dave”, participant_id: 001, birth_date: 4/13/2000, height: 21, weight: 22}
    - Invalid form inputs:
        - { name: “Dave”, participant_id: 001, birth_date: current_date, height: 21, weight: 22}
        - { name: “Dave”, participant_id: 001, birth_date: 4/13/2000, height: -21, weight: 22}
        - { name: “Dave”, participant_id: 001, birth_date: 4/13/2000, height: 21, weight: -22}
        - { name: “Dave”, participant_id: 001, birth_date: 4/13/2222, height: 21, weight: 22}
    - Send form

2.1.6 ExportForm

This form allows users to specify the export conditions for a given study’s data set, specifically the exported file’s name and the date ranges that the researcher wants the study to pull from. We will be sending mock projects in Jest with start and end dates so that we can make sure the dates must be within the given project’s date range.

- **Equivalence partitions:** Valid names, invalid names, valid start dates, invalid start dates, valid end dates, invalid end dates
- **Boundary values**: Names with spaces, names without spaces, start dates before the project start date, start dates after or equal to the project start date, end dates before or equal to the project end date, end dates before the project end date, end dates before the export start date
- **Selected inputs:**
    - Load page
    - Load mock project data (start date 4/13/2020, end date 6/12/2020)
    - Valid form inputs:
        - { title: “export_1”, start_date: 4/14/2020, end_date: 4/15/2020)
    - Invalid form inputs:
        - { title: “export 1”, start_date: 4/14/2020, end_date: 4/15/2020)
        - { title: “export_1”, start_date: 4/12/2020, end_date: 4/15/2020)
        - { title: “export_1”, start_date: 4/14/2020, end_date: 6/20/2020)

        - { title: “export_1”, start_date: 4/17/2020, end_date: 4/15/2020)
    - Send form

2.1.7 LogIn

This form allows users to log in if they are not logged in, and gives them a log out button if they are logged in. We will be receiving the data from the login form and sending back a response with Jest to either say that the credentials were valid or invalid. There will also be a button for forgotten password, which will send an email to the mailing address associated with a username entered in the form.

- **Equivalence partitions:** Valid username, invalid username, valid password, invalid password, user logged in, user not logged in
- **Boundary values**: Usernames and passwords that our Jest mock indicate are valid, usernames and passwords that our Jest mock indicates are invalid.
- **Selected inputs:**
    - Load page
    - Valid form inputs:
        - {username: “valid_username”, password: “valid_password”}
        - {username: “valid_username”} and Forgot Password button clicked
    - Invalid form inputs:
        - {username: “invalid_username”, password: “valid_password”}
        - {username: “valid_username”, password: “invalid_password”}
    - Send form
    - Click logout button

2.1.8 StudiesList

This component displays a list of studies that the current user can access. We will use Jest to mock several users being logged in as well as a list of studies they can access.

- **Equivalence partitions:** User logged in, user not logged in
- **Boundary values**: User logged in and able to access at least one study, user logged in and not able to access any studies, user not logged in
- **Selected inputs:**
    - Load page
    - Get studies user can access from Jest mock

    - Click study in list to access study

2.1.9 Study

This component displays the details of a specific study if a user can access it. We will be sending study details using a Jest mock, as well as current user details also using a Jest mock.

- **Equivalence partitions:** User logged in, user not logged in
- **Boundary values**: User logged in and able to access at least one study, user logged in and not able to access any studies, user not logged in
- **Selected inputs:**
    - Load page
    - Get study data from Jest mock
    - Click enroll link to get to EnrollParticipants

2.1.9 CreateUser

This form allows users to create a new user with a username, email, and password. We will be sending values to a Jest mock which will then send back a response to indicate if the credentials were valid or not, which will be displayed to the user.

- **Equivalence partitions:** User logged in, User not logged in, Valid username, invalid username, valid email, invalid email
- **Boundary values**: Username with spaces, username without spaces, email with a correct address format, email without a correct address format
- **Selected inputs:**
    - Load page
    - Valid form inputs:
        - {username: “user_name”, password: “password”, email: “email@email.com”}
    - Invalid form inputs:
        - {username: “user name”, password: “password”, email: “email@email.com”}
        - {username: “user_name”, password: “password”, email: “email@email”}
        - {username: “user_name”, password: “password”, email: “email”}
    - Send form

2.1.10 UserList

This component displays a list of users to the superuser. We will use Jest to mock several users being logged in as well as a list of studies they can access.

- **Equivalence partitions:** Superuser logged in, superuser not logged in
- **Boundary values**: Superuser logged in, superuser not logged in
- **Selected inputs:**
    - Load page
    - Get users from Jest mock
    - Click user in list to edit user

2.1.11 EditUser

This form allows superusers to edit a user's information in a form similar to that in CreateUser. We will be sending values to a Jest mock which will then send back a response to indicate if the credentials were valid or not, which will be displayed to the user.

- **Equivalence partitions:** Superuser logged in, superuser not logged in, valid username, invalid username, valid email, invalid email
- **Boundary values**: Username with spaces, username without spaces, email with a correct address format, email without a correct address format, superuser logged in, superuser not logged in
- **Selected inputs:**
    - Load page
    - Valid form inputs:
        - {username: "user_name", password: "password", email: "email@email.com"}
    - Invalid form inputs:
        - {username: "user name", password: "password", email: "email@email.com"}
        - {username: "user_name", password: "password", email: "email@email"}
        - {username: "user_name", password: "password", email: "email"}
    - Send form

2.2 Express Server

Our express server uses an express object to listen on a chosen port for incoming http requests. When a request is received it is validated and sanitized by our custom functions before being passed on to the designated route handler. The route handler will then request the corresponding data from the database and return a result.

Because most of our functions only communicate with other systems the only key internal functions to test are the request validation and sanitization functions.

2.2.1 sanitizeRequest

We will test this function by creating mock request objects with the required fields and passing these objects to the sanitization function. Then we will pass the result to a mock next() function and compare the result to the expected result using Jest.

- **Equivalence partitions:** valid request body fields, invalid request body fields.
- **Boundary values**: null, example known sql code/escape sequences with spaces, pure alpha characters no spaces, alpha characters with dashes and underscores
- **Selected inputs:**
    - invalid: { “route”, "hostname", "session", “body”: {"name": "test_study_20", "study_id": "'Robert'); DROP TABLE STUDENTS; --')"} }
    - valid: { “route”, "hostname", "session", “body”: {"name": "test_study_20", "study_id": "405"} }

2.2.2 checkRequired

CheckRequired takes two inputs, a template object created by the route handler, and an http request object. This function will be tested by creating a mock request object using Jest and passing it and the template from each route. The request objects needed will be simpler than sanitizeRequest because the function is only designed to make sure the request contains the required fields and that those fields are not null.

- **Equivalence partitions:** missing fields, complete fields, extra fields
- **Boundary values**: null, all required body fields except one (randomized), all body fields, and extra body fields
- **Selected inputs:** {}, { “route”, "hostname", "session" }, { “route”, "hostname", "body", "session" }, { “route”, "hostname", "body", "session", “surprise”, “extra” }

2.3 Export Support Program

Our export support program utilizes two main methods for export management. These methods are implemented using express routes and return data as objects. The first method is createExport and it uses data from a sanitized request to create an export object using the class ExportParams. The second method, getExportStatus searches for an existing ExportParams object matching the fields from a request and if it exists calls that object's getStatus method returning the result as the body of the response.

2.3.1 createExport

The create export function accepts an http request and response object as arguments. Our plan to test this function using unit tests is to create mock request and response objects using Jest to simulate an actual export request. Our tests will check whether the returned object is a null value or an ExportParams object and if so, will validate using the unit tests defined for all ExportParam objects.

- **Equivalence partitions:** complete request body fields, missing request body fields.
- **Boundary values**: null, missing one random required field, only required fields, all fields
- **Selected inputs:**
    - Invalid: { "name": "test_study_20" }
    - Valid: { "name": "test_study_20", "study_id": "405" }

2.3.2 getExportStatus

GetExportStatus accepts an http request object with information about a desired export, the main job of this function is to search the export queue to find a matching study export and return the status of that export. If there is no data matching the headers in the request then the function will respond by sending the appropriate http error status. Our testing strategy for this function is to create an example export array and example http request objects using Jest and passing these values to the function.

- **Equivalence partitions:** complete request body fields, missing request body fields.
- **Boundary values**: null, missing one random required field, only required fields, all fields
- **Selected inputs:**
    - Invalid: { "name": "test_study_20" }
    - Valid: { "name": "test_study_20", "study_id": "405" }

2.4 Fitbit Support Program

FitbitSupportProgram is the part of our system responsible for requesting data on study participants from the official fitbit API. When new data is available the program will make the necessary requests and save the result to the database. Additionally, after a new participant is added the program will request the historical data from fitbit needed to backfill the data needed for the study. To test this system we will focus on the three main isolated functions getToken, getNewData, and backfillData using Jest to create a test participant pool and session objects.

2.4.1 getToken

The getToken function takes a session object containing a OAuth token, refresh token, and time to live value, and a time object representing when the session was issued. The function will first check if the session exists and is still valid before sending a request for a new or refresh token if needed. Testing this will require us to create example session objects, pass them to the function and check the expected against the actual result.

- **Equivalence partitions:** complete session fields, null session, missing or additional fields.
- **Boundary values**: null, missing one random required field, , complete fields
- **Selected inputs:**
    - invalid: { "token", "refresh", “ttl” }
    - Valid: {null},{ "token", "refresh", “ttl”, “issued” }

2.4.2 getNewData

The getNewData function gets data from the Fitbit API whenever new data is available at regular intervals. We will test this by using Jest to mock Fitbit data being sent to the function.

- **Equivalence partitions:** complete fitbit data, incomplete fitbit data, null data
- **Boundary values**: null, missing one random required field, complete fields
- **Selected inputs:** Examples of Fitbit data as provided by the Fitbit API, both valid and invalid based on the boundary values.

2.4.3 backfillData

The backfillData function is called when data is missing from the database. We will test this by once again using Jest to mock Fitbit data being sent to the function.

- **Equivalence partitions:** complete fitbit data, incomplete fitbit data, null data, date where data is missing
- **Boundary values**: null, missing one random required field, complete fields, no date, one date
- **Selected inputs:** Examples of Fitbit data as provided by the Fitbit API, both valid and invalid based on the boundary values.

3. Integration Testing

Once unit testing has completed testing all the modules for their intended tasks, we move on to integration testing as it is the process of testing the interface between modules. This means it needs to test the modules to expose the faults of the interactions between each module. This part of testing is important because for WearWare we need consistent and accurate data that can be used for our client and other researchers. For this we will need to test the communication between the Web App, Request Server, Database, and FitBit API as shown in the below diagram.

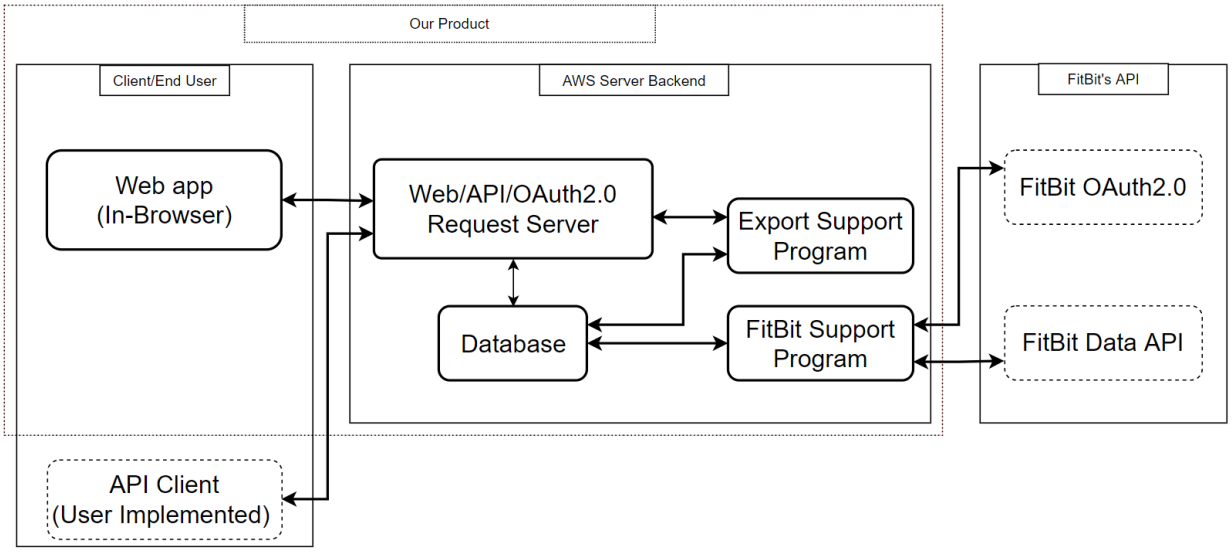

Figure 3.0 WearWare Architecture Diagram

3.1 Web App to Request Server

We will be using the Vue testing tool Cypress for testing our web app and request server integration. Unlike Jest which is built for individual units, Cypress is designed for end-to-end testing, which makes it much better for our integration tests. For this project, we will be writing tests for all of our units like we did for our unit testing. However, instead of using Jest mocks to simulate the backend responses, we will instead run these tests while the web app is fully connected to the request server. This will also serve to test the connection between the request server and the database, since the webapp will rely on data from the request server and database.

3.2 Request Server to Database

The communication between the request server and the database will be tested using Jest. Jest will be used to send queries to the database via requests to the request server for specific data. The data will then be analyzed to see if it accurately matches the data we are looking for. We will include tests for every kind of entry in the database to ensure that every single type of database objects can be properly extracted and added via our request server.

3.3 Database to Fitbit API

The communication between the database and Fitbit API will be tested using Jest. Jest will send mock data to the Fitbit API. The data from the Fitbit API will be sent back to the database. The data will be analyzed to see if it sent properly without any errors.

4. Usability Testing

Now that we have established our unit testing and integration testing plans, we can move away from testing code and toward testing our user interface, specifically with usability testing. Usability testing is a type of testing that is not based on the code of our program but instead on making sure that users can use our program without having any confusion or issues with the workflow. This is very important, as a fully functional and bug-free program is useless its end users are not able to actually use it. As a result, we must find ways to see what the perspectives of potential users are, which is what this section will be focused on.

As stated in the introduction, the main target audience for this project will be health researchers who want to conduct studies using Fitbits to gather health data from study participants. As a result, this audience will not necessarily be technically literate; their speciality is health, not technology. Additionally, if they end up causing problems accidentally due to their lack of tech literacy, they could end up losing valuable health data that could have been used to conduct studies and save lives. Because of this, we must ensure that our project is usable for people who are not technically literate, preferably those with no background in programming or computer science unlike us. This naturally means that we cannot test it ourselves; rather, we plan to use focus groups as recommended to us by our client. These focus groups will be mainly composed of contacts that our client has who work in the health research field and the team's peers who are not technically literate. Since these groups have been recommended by our client, who works in the health research field, we believe that they will be sufficient to fit our needs for usability testing.

Focus group tests will be conducted during scheduled weekly meetings either in our lab or remotely online via Zoom. Our peer groups will be composed of four peers, one from each team member, while our health researcher focus group's size will entirely depend on how many people our client can get in contact with. We will be meeting with each of the two groups separately each week to view how they interact with the project and see what difficulties or misunderstandings they have while using it. After each test we will also provide a feedback survey for our focus group participants to fill out. This will have questions including the following:

- On a scale of 1 to 10, how would you rate the experience of using this web app?
- Please explain any difficulties you had with the web app.
- Is there anything that you feel is missing that we should add?
- Is there anything that seems excessive that we should remove?

Using this feedback and any notes that we take while observing the focus group participants, we should be able to get a good idea of what issues there are with the user experience. At the moment, we have about three weeks left for this project - as a result, our timeline will be somewhat tight. However, since these meetings will be conducted

twice a week, once with each group, and we will also be able to test the user experience ourselves and with our client to even further refine our knowledge on any issues with it and how to fix those issues, we believe that we will be able to iron out any potential issues and thus deliver a high quality and easy to use product to our client.

5. Conclusion

Health research is very important, and if used correctly our project could help save lives by enabling researchers to use Fitbits instead of the current expensive tools that they use. However, this will only actually help researchers if it works properly without any issues that could hinder its use. In order to prevent these issues, we need to do software testing on our product to help catch these issues before our client encounters them. Software testing is a very important part of the development process, as we need to ensure that everything works properly before we give our product to our client. If we delivered a buggy product that did not work as intended, it would prevent its proper use and thus mean our project would fall short of acceptability.

We are currently planning to use three different approaches to test our project: unit testing, integration testing, and usability testing. Unit testing will act as a foundation for our testing plan, letting us test the individual “building blocks” that we are using to build our product and thus allowing us to ensure that each piece lacks issues before they are all put together. Once that is finished, we will be able to do integration testing, which looks at how these individual parts all work together and interact with each other. This will help us to get a more comprehensive view of the project as a whole, and find any issues that our individual parts had that we may have overlooked. At the same time, we will also be doing usability testing, which will allow us to see how users interact with our system and thus find out which parts of the user interface could be improved and what parts are confusing or clunky. This will ensure that our client will be able to use the system properly without any complications caused by problems with the user interface.

At the moment, we are finalizing our project and thus will be able to start working on unit testing before we do integration testing. At the same time, we are assembling our focus groups so that we can begin usability testing, which will start very soon. With all of our plans in place for testing, we can rest assured that we will find any issues with our project and fix them before they become a real problem, and thus deliver an error-free and easy to use product to our client that can change the field of health research and help save lives.