Software Testing Plan

Version 2.0

March 26, 2021

Project Sponsor: Chris Doughty
Team Mentor: Andrew Abraham
Team Members: Kainoa Boyce, McKenna Chun, Gregory Geary, and Wesley Smythe

Table of Contents

Figure 1: Frogs are one of the many creatures that make up the biodiversity of Earth.[1]

1 https://royalsociety.org/topics-policy/projects/biodiversity/

1 Introduction

The Madingley Model is a revolutionary biodiversity and ecosystem model that has the capability to generate data over wide areas of land, limited only by the maximum surface area of Earth. Another unique feature of this model is that it is able to include data for both oceanic and terrestrial areas. In addition to this, it is able to factor in a multitude of different scenarios over these selected areas, such as human driven ones like climate change or deforestation.

The model itself was created with the intent of providing policy makers accurate biodiversity data, of which they could then use to influence the types of policies they put into place. However, due to the underlying fact that nature is an incredibly complex system, the data sets generated by this model are extremely large and very difficult to manipulate and interpret for the average human being.

To solve this, Biosphere was developed as a progressive web application that allows the user to view different summaries of Madingley Model data in a cost effective, easy, and efficient manner, requiring little to no prior knowledge on biodiversity and ecosystems. The application starts by prompting the user to input various data (user-type, location, scenario, and intensity of that scenario), these are then used to determine the specificity of the scenario data, where the desired scenario should be run, and the intensity of that desired scenario. From there, the application will fetch the corresponding Madingley Model data to be displayed via a heatmap, line graph, and bar graph.

Now that the app has been built out, it is imperative to check that all key components are up to par, and allow the users to effectively fulfill the main purpose of the Biosphere Application. To ensure a healthy user experience, the application undergoes a series of different application tests, these are:

- Unit Testing
- Integration Testing
- Usability Testing

The first type, unit testing, will be conducted to ensure that each of the individual components of the app are working properly. This would consist of running trials on our functions that perform tasks, examples of these tests include, but are not limited to:

- Making sure data is being saved upon user input properly such as the user's location or selection of scenario and user types
- Ensuring that the correct data is being pulled and processed by the backend

Secondly, integration testing will be conducted to ensure the key components of the app are all working together properly. This type of testing for our application consists of verifying correct responses between multiple modules:

- Ensuring that the API Gateway is correctly sending requests, and receiving responses between the back-end and front-end modules
- Confirming that different front-end components communicate with each other to provide a healthy user interface to the user
- Confirming that the format of data is consistent throughout the entire application to create a modular environment for a diverse range of situations

The last type, usability testing, shifts the focus away from the functionality of the technical components themselves, and more towards how users interact with these components. This consists of tasks such as:

- Finding a wide variety of different users that would have different experiences and access the application differently
- Getting feedback from these various users that have tried the application in order to ensure that we have a great user experience, and intuitive user interface
- Analyzing feedback to make changes and prioritizing specific tasks based on importance to the user's experiences

All of the listed tests above constitute a better application by allowing us as the developers to maintain a consistent standard of performance for the product. They also allow us to find problems and errors that may occur only in specific circumstances that were not clear or apparent to us before. This applies for all of the different testing methods, for example, extensive unit tests allow for a wide range of situations, whereas usability testing allows for us to see how the users would really interact. Therefore, each test plays a key role in identifying critical issues that we may or may not have and are extremely important in the development of our Biosphere application.

2 Unit Testing

Unit testing is used to break large pieces of software into individual components or functions. This aims to ensure that each unit is performing as expected. Unit testing can be done automatically or manually. In order for unit testing to be validated, the developer or test engineer must generate tests, and define their outputs. Once numerous tests have been generated that cover a wide array of possible inputs they are tested, and their results are generated. If there are failures in unit testing, it is usually caused by a logical disconnect between the code written and the intended output. The secondary purpose of unit testing is to find obvious issues like, unrestricted access to various parts of a system, or the mishandling of control or special characters. An example of this would be an SQL injection as a result of improper handling of the input, and allow it to run as code, rather than constricting it to a string datatype.

2.1 Back-end Testing

The back-end can be broken down into three major components: data storage and retrieval, data processing, and the back-end interface. Each unit will be tested extensively with each unit containing its own criteria.

Unit Test	Description	Boundary Values	Example Input	Expected Response

API: Single Valid	Generate a single valid response that is then passed through the API Gateway.	distance values cannot exceed 800km.	min_distance=0
			max_distance=200000
				{
				“Next”: None
				}

API: Onion Valid	Generate an onion handled response that is then passed through the API Gateway.	distance values must be greater than 800km	min_distance= 2345
			max_distane = 340953
				{
				“Next”: event
				}

Data Retrieval: Non-Existent File	Create a data request for a file that clearly does not exist.	A string that looks like a file i.e. some_dir/some_file .csv	get_object(file=”fake.csv”)	FileNotFoundError
Data Retrieval: Out-of-Bounds Request	Create a data request for a file that is located in a different file system	A file that exists. The file path must start from the root.	get_object(Bucket=wrong_bucket, file=”file.csv”)	FileNotFoundError ForbiddenAccessError
Data Validation: Correct Data Types	Create a request with a valid size and type to be tested against validation library	The request must conform to the restrictions of the requested input	{ “user_type”: public … scenario: “CLIMATE” }	“[DEBUG]: Validation: Passed”

Data Validation: Incorrect Data Types	Create a request with a valid size and type to be tested against validation library	The request must be incorrect within the constructs of the validation functions	{“fish”: “taco42”}	“[DEBUG]: Validation: Failed”
Data Processing: No Returned Data	A data request using semi-legitimate parameters. For a dataset that does not exist.	The request must be approved by validation functions	{ “user_type”: public … scenario: “CLIMATE” }	{“statusCode”: 400, “body”: None}
Data Processing: Improper File Format	A data request using legitimate parameters for a file that is mislabeled, or of an invalid type	The request must be approved by validation functions	{ “user_type”: public … scenario: “CLIMATE” }	{“statusCode”: 400, “body”: None}
Data Processing: Invalid Values	A data request using valid parameters, for a non-numeric value	The request must be approved by validation functions	{ “user_type”: public … scenario: “CLIMATE” }	{“statusCode”: 400, “body”: None}

Table 1: Back-end Unit Testing Criteria

Finally, for the purposes of security, the back-end components will be compared against the Common Vulnerabilities and Exposures (CVE) database to ensure that most known vulnerabilities have been patched. According to preliminary research there are 115 vulnerabilities related to AWS and 494 vulnerabilities linked to Python with the most recent entry dated for March 4, 2021. It should be noted that not all vulnerabilities will be related to this application, and some may be illegal to test without a Certified Ethical Hacker (CEH) degree or being contracted as a bug or vulnerability finder.

2.2 Front-end Testing

The front-end, similarly to the back-end, can be broken down into specific modules that each perform a task that they are specialized for. While there are many smaller modules included in our front-end, some major modules that are prone to invalid responses include: location selection and data retrieval. Like the back-end, each unit is tested for a variety of scenarios, both successful and not.

Unit Test	Description	Boundary Values	Example Input	Expected Response
Location Selection: Map Input	Receive input from Google Maps API selection, and send a data request to back-end	Latitude: [-90,90] Longitude: [-180,180] Radius: [0, 12000]	Latitude: 52 Longitude: 105 Radius: 1240	Request made to Back-end
Location Selection: Valid Manual Input	Receive manual location input, and send a data request to back-end	Latitude: [-90,90] Longitude: [-180,180] Radius: [0, 12000]	Latitude: 40 Longitude: 105 Radius: 2421	Request made to Back-end
Location Selection: Invalid Manual Input	Receive an invalid manual location input and produce an error response	Latitude: [-90,90] Longitude: [-180,180] Radius: [0, 12000]	Latitude: -105 Longitude: 185 Radius: -2752	Error: Invalid Input
Data Retrieval: Successful Data Retrieval	Request and retrieve the correct files selected from the location and scenario options	Value 1: 0 <= Val1 < Val2 Value 2: Val1 < Val2 <= 1625	generateRequest (0, 249)	Madingley Onion Data, statusCode: 200
Data Retrieval: Failed Data Retrieval	Failed request of correct data and display an error	Value 1: 0 <= Val1 < Val2 Value 2: Val1 < Val2 <= 1625	generateRequest (0, -515)	No Data, statusCode: 400

Table 2: Front-end Unit Testing Criteria

3 Integration Testing

In addition to unit testing, another form of testing is integration testing. Even though integration testing might seem very similar to other testing types, it has one main application that separates it from other tests. Integration testing focuses on how different components of a program work together. If we use a car as an example, integration testing would be used to test if the motor, the frame, and the tires all work together properly. If the motor runs, but it doesn't connect to the tires properly, the car isn't going to move. To get the car to move, their individual functions have to work together seamlessly. The same seamless interaction of different car parts needs to occur with the different components of our program. In particular, we need the data storage, data visualization, API Gateway, and the Lambda functions to all communicate effectively with one another. If they can all work together perfectly, we can produce the overall result that we want to achieve.

3.1 Back-end: Data Storage and Retrieval

A key component of the back-end involves requesting, and retrieving data files from the pre-specified file system. This is done through the lambda function that is given special, and limited privileges in order to promote the principle of isolation, and least privilege. In order to test the scope, and limitation of this aspect integration test will be done.

Integration Test	Description	Expected Response
Perform a valid action specified by the Lambda's IAM policy	A request will be generated that is within the scope of the predefined policy	This test will be given a passing mark if it performs the action and does not return AccessDenied.

Table 3: Back-end Data Storage and Retrieval Integration Test Table

3.2 Back-end: Data Validation and Interpretation

Once the data is retrieved from the file storage system, it must be validated and translated to a valid data type that can then be handled by the remaining aspects of the back-end.

Integration Test	Description	Expected Response
Take a valid data file and check its internal values.	Take a valid data file as an input, then check the internal values for column names, length, and cell type.	If this action is performed correctly then no errors should be returned, since cell values and column indices are hard-coded into the back-end.
Take a valid data file and transform it into a non-networked data type.	Take a valid data file then perform the static and dynamic operations translating it from bytes to JSON.	If the actions are performed correctly then a populated dictionary object will be returned.

Table 4: Back-end Data Validation and Interpretation Test Table

3.3 Back-end: API Gateway and Lambda

In order to communicate between the frontend and backend, a gateway is used which acts as a trigger to the back-end components. Once the lambda function is triggered it performs a pre-specified task then returns numerous headers outlined by AWS. These headers are then parsed by the frontend to be parsed and visualized.

Integration Test	Description	Expected Response
Generate an API Request and wait for the output.	Use the Live API Gateway or API Sim to pass a request to the lambda function to then be called, and data returned.	If the request is valid, and does exist, then it should return a response with either a statusCode of 200 or 400.

Table 5: Back-end: API Gateway and Lambda Test Table

3.4 Front-end: Location/Scenario Selection and Data Requests

The location selection module must pass in three valid numbers for the latitude, longitude, and radius of the circle in which the data is retrieved for. The values are then stored, and once the user selects their user type and scenario options, the API Gateway creates a request for the back-end for the appropriate data.

Integration Test	Description	Expected Response
Collection of scenario and location data, then request data.	Different scenario data is selected by the user and stored, then used to request specific sets of data from the back-end via the API Gateway.	A successful response will return no errors. Wrongly formatted requests will produce errors. Invalid input will cause errors elsewhere.

Table 6: Location/Scenario Selection and Data Requests Test Table

3.5 Front-end: Data Retrieval and Data Visualization

The API Gateway is used to retrieve data from the back-end that was previously requested using the parameters selected by the user. This data if successfully retrieved is then used to dynamically generate graphics such as buttons, charts, and a heat map produced by the Google Maps API, along with an appropriate legend color scale.

Integration Test	Description	Expected Response
Retrieval of JSON data used to create visuals and other UI components.	The API Gateway gets the previously requested data that the back-end has parsed and allocated for use. This data is then retrieved and used by the front-end to visualize the datasets using heatmaps, graphs, and other graphics.	If successful a series of requests will return a finite amount of data and a statusCode of 200. The graphics and other UI components will then be generated based on this input. Otherwise, it will return no data and a possible statusCode of 400.

Table 7: Data Retrieval and Data Visualization Test Table

4 Usability Testing

Besides unit testing and integration testing, a third type that we will be using is usability testing. The purpose of usability testing is to test the interactions between the application and the target audience. This type of testing focuses on the overall quality, and intuitiveness of the application. It will simulate what a typical user will do in the app after it is deployed. We will specifically be analyzing the speed of the app, the time it takes to get over the learning curve in the app, and whether the app can be changed to make it more user friendly.

Usability testing is extremely important for the end-user facing aspects of the application. In particular, the users should be able to use the app without any outside assistance from a team member. If they can't navigate the app by themselves, we will need to change something because we won't always be on the app to offer assistance. Also, this is one of the first times that someone outside of our team will be using the app, so it will be a good way to get constructive feedback from others. For example, we might realize that there are parts of the app that we overlooked as easy to understand or areas where there isn't enough instruction for the user.

Since our application is a Progressive Web Application (PWA), we are hoping to get feedback for the web version, the iOS version, and the Android version of the app. By creating one code base for all of the versions, it should transfer to each device correctly. However, there could be problems or areas for improvement that only show up on one type of device. It is important we check each version of the app. For example, we might find that a smaller screen introduces new problems that aren't present in the web version. Whatever device the user is using, we will have one primary method for retrieving testing information from the user.

In order to get the best feedback, our team will be using zoom to visualize the user's screen and get their input on certain areas of our application. Most of the testing will be taking notes on the user's interactions with our app, but we hope to also get some feedback from the user about their experience. To accomplish this, we will give them a very vague task to complete within the app. Then, we will watch as they progress through the app. We will document specific information on how long it takes for the user to get to the destination and if there are any spots where they were confused on what to do next. When possible, we will get several people from the same user type to test out the application at the same time. By doing this, we hope that it will cultivate useful discussion and questions while they are going through the application. While on zoom we will try to tailor the testing to the user's specific background.

Since we have three very different user types, it will also be important to see if we addressed each of their needs. To begin, we are assuming that the general users haven't heard about the Madingley Model prior to our testing meeting. As a result, we need to make sure they don't get lost while navigating the app. Both the scientists and policy makers will typically have a detailed scientific background. It will be important for us to provide them a version of the app with this in mind. We don't want them to feel bored because we added too many useless things. It is important that we can test each of the user versions because they all produce different variations of the data. For example, the general user has the option to select 4 different output scenarios, but the scientist will have the option to select from 20 different variables. Therefore, if we only test the app from the general user's perspective, we will be missing the other ⅔ of our application. It will be important to test every part of the app that any user could interact with. In the case of our application, the user interacts with the: geolocation module, visualization module, and the user interface.

4.1 User Interaction

4.1.1 Geolocation Module

In this module, there are three main pages we will need to test. First, we will need to test the select location map page. On this page, we need to make sure that the user can intuitively navigate the map to select their desired location. Also, we need to test whether the user knows that they can change the radius of the circle. Second, we need to see if the user has any problems with the manually input coordinates page. On this page, we investigate if the user has any problems entering the latitude, longitude, and radius. The third geolocation check we need to test for is when the user selects to use their current location. The program needs to correctly retrieve the device's location and then properly display it on the map page. In the process, it should also alert the user if the location services are turned off in their browser or computer.

4.1.2 Visualization Module

For this module there is only one page to test, but there is a lot of important information that could be wrong. First, we need to make sure that the user can correctly export a pdf of the results. There is an export PDF button within the page. We need to verify the user can spot the button and that they know to click the button. Second, the data should be displayed in a way that makes sense for the user. If there is too much information on one page, it might confuse the user. If there is too little information on the page, the user might not see a point in even using our app in the first place. Lastly, the team also needs to test if the table and map are readable. In other words, we want to find out if the user can interpret the data in these areas.

4.1.3 User Interface

One of the most important UI tests for our team is checking if the user can correctly navigate from one page to the next. For example, the user should be able to submit the location (on the map page) and move onto the scenario option page without any problems. Even though we don't have all the languages implemented yet, it will also be beneficial to test out the translation feature. This will show us English text that we might have forgotten to translate or text that is incorrectly translated.

4.1.4 Back-End: Lambda Functions, API Gateway, and S3 Bucket

Given that the end-user does not natively interact with the back-end, this module can be excluded from usability testing. We specifically designed our app to hide these aspects from the user. If they had access to these parts, they could change the data or even destroy the inner workings of our application.

4.2 Testing Regime

4.2.1 Quantitative Description

In order to get a wide variety of feedback, our team is hoping to test a large sample of people. Ideally, we would also want a similar number of users from each of the three user types (general user, scientist, and policymaker) to test out our application. However, we know that finding someone who falls under the policymaker category will be harder than finding someone who qualifies as a general user. Therefore, our goal is to have at least three different groups of users test out the general user functionality, at least two groups of users who can test out the scientist functionality, and at least one test from a policymaker. By testing them in groups, we should get better communication and it will hopefully result in more detailed suggestions. We are hoping that by getting feedback from more than two groups for each user type, we can limit the feedback based on one person's prior experiences or personal bias.

4.2.2 User Studies and Acceptance Testing

Since our team has three different user groups, we are hoping to tailor some user stories specifically for each group. Some of these user type dependent tests can be seen in tables 9-11. Overall, tables 8-11 show that each task will correspond to one of the modules mentioned previously and list out several factors to determine if the task was successfully completed or if it needs to be refined.

Examples of Tests for All Users			
Tasks	Associated Module	Acceptance Testing	Testing Results (up to March 19)
The user should be able to create a PDF of the results	4.1.1 4.1.2 4.1.3	Success: ● If the user can correctly navigate the app and get a PDF using the button on the results page Needs Work: ● If the user can't get to the results without assistance ● If the user uses the browser to create a PDF instead of the provided button	**Success** 5/5 groups were able to create a PDF. However, ⅗ groups had at least one person who took longer to find the button than we would like.
The user should be able to start a new simulation and get the results for ____ (insert a mixture of countries all over the world)	4.1.1 4.1.2 4.1.3	Success: ● If the user can get results for the specified country. ● If the user can start a new simulation and it displays new information from before Needs Work: ● If the user can't start a new simulation without assistance ● If the user gets results for the wrong country	**Success** 5/5 groups were able to move the circle to the specified country and get results for that specific region.
The user should be able to use these latitude and longitude coordinates to find the corresponding results	4.1.1 4.1.2 4.1.3	*Send latitude and longitude coordinates to the user in the zoom chat* Success: ● If the user can get results for their specified location ● If the results page shows a circle with the specified radius Needs Work: ● If the user doesn't know where to input the values without assistance ● If the user gets an error or the results don't populate correctly	**Success** All the tested groups* were able to get the results matching the latitude and longitude coordinates that were entered. *only tested on ⅗ groups.
The user should be able to walk through the app in the French language	4.1.3	Success: ● If the user understands what all the text means in the French language. Needs Work: ● If the user has trouble navigating the app because they don't understand the text without assistance ● If the user uses Google Translate outside of our app	**Success** Camille from scientist group 1 was able to navigate the app.

The user should be able to explain what the Madingley Model is and explain what each of the scenario options mean	4.1.3	Success: ● If the user got a good understanding of the app from the About page ● If the user correctly summarizes the main points of the Madingley Model and the scenario variables Needs Work: ● If the user doesn't understand what the Madingley Model is without Googling it. ● If the user is just clicking buttons and doesn't understand what the scenarios mean	**Needs work** ⅖ groups (which had no previous experience with the Madingley Model) had a hard time understanding that the results were based on the Madingley Model. The getting started page should be edited to include some Madingley information

Table 8: User Study for All User Types

Examples of Tests Specifically for General Users			
Task	Associated Module	Acceptance Testing	Testing Results (up to March 19)
A general user should be able to easily select a new output variable (out of the 4 possible options)	4.1.2 4.1.3	Success: ● If the general user can select a new variable and the map refreshes to display this new information Needs Work: ● If the general user doesn't realize that they can select a new output variable without assistance	**Success** 5/5 groups were able to change the variable from the current default of “allelic diversity” to any of the other output variables.
A general user should be able to describe what each of the output variables mean	4.1.2 4.1.3	Success: ● If the general user can define what each of the output variables mean Needs Work: ● If the general user has no idea what they are clicking.	**Needs Work** 3/3 of the general user groups had a hard time understanding what the variables meant. Suggestion: add descriptions next to the variable name

Table 9: User Study for General Users

Examples of Tests Specifically for Scientists			
Task	Associated Module	Acceptance Testing	Testing Results (up to March 19)
A scientist should be able to easily interpret the results shown on the results page	4.1.2 4.1.3	Success: ● If the scientist can explain what the results are showing Needs Work: ● If the scientist doesn't know what is being displayed on the page. ● If the scientist is overwhelmed with all the information being displayed	**Needs work** Scientists weren't able to get the results to display. The users were stuck on the "Madingley Data loading…" notification.
A scientist should be able to select between the 20 raw data output variables	4.1.3	Success: ● If the scientist can switch from 1 output variable to another one and the results change as a result Needs Work: ● If the scientist doesn't know that they can select one of the other 19 output variables ● If the scientist gets the same map even after they individually select several of the other variables	**Needs work** Scientists weren't able to get the results to display. The users were stuck on the "Madingley Data loading…" notification.

Table 10: User Study for Scientists

Examples of Tests Specifically for Policymaker *Testing results for this user type is planned for March 22*		
Task	Associated Module	Acceptance Testing
The policymaker should know that the variables on the results page are calculated EBV values. Test by asking the policymaker what each of the variables represent	4.1.2	Success: ● If the policymaker can identify that the variables are calculated based on their selection in the app Needs Work: ● If the policymaker thinks that the variables are hardcoded values

The policymaker should be able to get results in a reasonable amount of time or get a warning that it might take a long period of time to retrieve the requested data	4.1.3	<u>Success:</u> ● If the policymaker gets results in less than 2 minutes. ● If the policymaker gets an alert that it might take a long time to process their results <u>Needs Work:</u> ● If the policymaker thinks the app is broken because the app is on the "Loading Madingley Data" page for too long

Table 11: User Study for Policymakers

Some of the follow-up questions that we asked throughout our current usability testing:

1. I noticed you struggled with _____ part, do you have any suggestions on how we can improve that for other users?
2. In the beginning, did you feel that there was enough background information to understand the key parts of the app?
    a. Did you get a good understanding of the app's purpose?
3. What are some other features you might like to see in this application?
4. Can you summarize what the line plot and bar graphs on the results page are saying?
5. Does the button layout make sense?
    a. Was it easy to know that they were buttons and not just text?
6. Can you tell me what each page does?
7. Can you easily find the main operation of the application?

Specific areas that we took notes on while we were analyzing the screens of the different users:

1. Did the user have any problems scrolling up and down the page?
2. Were there any pages of the application where the user spent too much time trying to figure out what to do next?
3. Were there any problems with the user getting their current location?

4.3 Testing Timeline

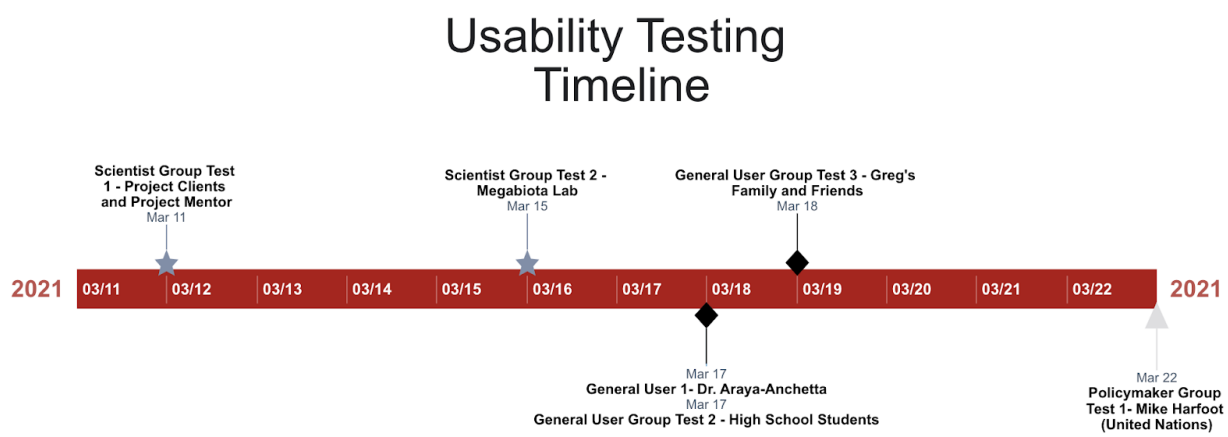

Figure 2: Usability Testing Timeline between March 11 and March 22

The team was able to complete about 80% of the usability testing during the week of March 15. However, we still have some last-minute testing that will take place during the week of 3/22. The first test the team did involved Roo, Camille, and Dr. Doughty on 3/11. This group of individuals all work with the Megabiota lab at NAU, so we classified them as scientist group number one. When it was Camille's turn to test out the application, we had him use the French translation so he could give us feedback on the French version of the app.

Next, Kainoa met with the rest of the Megabiota lab to get feedback from group two of scientists on 3/15. Then, Wesley held a zoom meeting with Dr. Araya-Anchetta (Dr. A) on 3/17. Dr. A is an introductory biology and genetics professor at NAU. Since Dr. A doesn't specialize in biodiversity, she was considered our first general user group. We considered her to be a general user who has a large variety of scientific experience. This gave us a different perspective from someone who knows less about scientific topics. Wesley then met with another general user group on 3/17. This group consisted of three high school aged students. We aren't sure if high school students will find this app useful, but we wanted to see if they could still understand the basic parts of the app. This group helped us find some technical assumptions that resulted in problems accessing current location.

Once those tests were completed, Greg met with another general user group on March 18. He got some insight from a member of his fraternity. This test helped us get feedback from a college student who isn't pursuing a Bachelor's in a STEM related field. Lastly, we hope to get at least one member from the United Nations to give us some feedback on our application. If all goes according to plan, we should be meeting with this policymaker during the week of 3/22.

5 Conclusion

Biodiversity and species richness are key measurements in ensuring the ecosystems around the world remain healthy. Currently, there is a strong decline in biodiversity and species richness. This is causing a degradation of ecosystem services like air filtration, water purification, and pollination. The depletion of biodiversity and these services will turn huge areas of earth into barren and uninhabitable wastelands.

In order to combat this decline, the Madingley model was developed to provide forecasts based on current and theoretical trends / scenarios that input biodiversity. The Madingley model is a genius piece of scientific software that requires a high level of expertise and computational power to operate in a reasonable amount of time. As a result of these limitations, there are very few people who can operate this model.

The goal of this project is to allow the target audience to have access to a predefined number of pre-run Madingley model scenarios. In order to reach the largest number of people, a progressive web application (PWA) is being developed. The application has several key modules including the: front-end user interface (UI), a cloud-based back-end used for data processing, and storage, a geolocation module for location based services, and most importantly a visualization module which showcases the data in a meaningful and easy to understand format.

As a result of these numerous modules, it is crucial to test these modules for: improper usage, security vulnerabilities, user experience (UX), edge cases and heavy or prolonged usage. In order to test all modules as thoroughly as possible this document outlines how the team will utilize: unit testing, integration testing, and usability testing amongst a wide array of users and environments to gain an understanding of the shortcomings of the application.

After some preliminary testing involving project sponsors, and various stakeholders some shortcomings were identified in the UX experience aspects, as well as the time-based restrictions caused by the sequential nature of the back-end. The UX and back-end efficiency will be areas that will have a stronger focus for the remainder of this development process with various iterations and testing done as improvements are made.

Given the rigor and continuous nature of the testing, the application delivered by this team will contain few to no instances of extreme lag from the back-end or an unclear user interface and experience.