



Software Testing Plan

Team Bird's iView

March 26th, 2021

Version 1.0

Project Sponsor: David Plemmons

Project Mentor: Sambashiva Reddy Kethireddy

Team Members:

Jonas Dunham Jordahl

Jordan Colebank

Chenhao Li

Tyler Riese

Table of Contents

Introduction	3
Unit Testing	4
Integration Testing	8
Usability Testing	11
Conclusion	23

Introduction

More than 50 million people spend more than \$40 billion every year on bird-related equipment and travel. Birding activities can bring people not only spiritual pleasure, but bring huge economic effects as well. Our client, David Plemmons, is an avid bird watcher and enthusiast. He has been working hard over the last three years to bring his vision to fruition. His product, the pEEp Smart Feeder, aims to get people more involved with the birds around them by providing an educational platform and vibrant community for people to connect and grow with other like-minded individuals.

The goal of our application is to build a web portal for users of the pEEp Smart Feeder, which will allow users to interact with their personal pEEP feeders as well as the greater bird watching community. On the pEEp website, bird watchers will be able to upload photographs of birds taken by the pEEp Smart Feeder's on-board camera, create posts on the pEEp community forum (optionally showing off their saved photos), and communicate with other bird watchers. Additionally, pEEp will provide learning tools so users can learn more about the birds they interact with.

When it comes to creating and designing software, there are a variety of ways to test your program. The three main testing methods we will be discussing are unit testing, integration testing, and usability testing, as well as two other techniques known as white-box testing and black-box testing. White-box testing is where the testers, otherwise known as the programmers, have knowledge of the actual code they are testing, whereas black-box testing is when the code is hidden from the tester. For our project, we will have our client conduct black-box testing of the pEEp web application to ensure the program works from a users' perspective, while the rest of the team will be conducting white-box testing.

We plan to implement unit, integration, and usability testing. Unit tests will test each part of our code to see if it behaves as expected and doesn't have unforeseen problems. We will write some unit tests for each module of our web application (i.e. home page, gallery, AI, etc.). The integration tests will examine how the various components of our project work together to produce the required functionality. In our case, integration testing will be done to ensure information is communicated correctly and consistently between the model, view, and controller modules. Usability testing will require people unfamiliar with our project to browse our mobile and web applications and complete various tasks to assess how user-friendly our applications are. This may also find any problems with our user interface and other software components. Usability testing is very important to our project because it ultimately simulates how end users interact with our product and will give us useful feedback on how to improve the product.

Unit Testing

Unit testing involves breaking down a program into smaller subsections, or units, that can be tested individually. Most commonly when creating software, it involves testing each function within the program. With unit testing, there is typically a three step process. The first step with unit testing is determining the initial module to be tested. Following that, the program is then “tested”, either manually by the programmer or with an automated testing framework. Lastly, the programmer then examines the output from the test.

By isolating each function of the program, testers can identify if each module of the program is accurate and working as expected. The goal of unit testing is to ensure that there are no unforeseen defects or problems within the program. The two ways programmers can conduct unit testing are with automated or manual testing. Automated testing involves the use of an automated tool that can produce multiple tests for a program in a timely manner. Manual testing involves the programmer following a test plan document while analyzing defects within the code without an automated tool. With both automated and manual testing, the programmer is required to write a test plan with detailed instructions for how they plan to test each function.

As stated above, unit tests involve breaking down a program into subsections to be tested. For our project, we will be conducting manual unit tests for both the user-interface of our program and the logic behind the program. Since we will be conducting manual unit testing, we will not be using any unit testing libraries or frameworks. The units for the logic behind the program will be the functions themselves. By doing a manual test for the logic, we will start by choosing a function to test and then write out a plan for how we want to test it. Our testing plan for each function will be structured by writing out a step by step process for a selected function, the number of input values necessary for testing, what values we will be inputting into the function, and, lastly, the expected results. These input values should contain both expected and unexpected values. For example, we have a function called `delete_picture`. The purpose of this function is to take in an image file and the folder the image is located in and remove it from the folder. The expected outputs for this function are either `True`, the image has been successfully deleted, or `False`, the image was not deleted.

To test this function specifically, the input values we should pass in would be multiple image files and folder paths and multiple input values that are neither image files or folder paths. The reason we want to test values that aren't the typical input values are because we do not want to return a response of “image has been successfully deleted” if the input values are incorrect. Additionally, if we pass in a parameter that is not

expected, it should return the correct boolean value of False. By doing this, we can determine if there are any unexpected defects within the function.

An example of how this function might be structured into our test document will be provided below.

Unit: delete_picture function

Boundary Values: image file and folder path

Standard Output Value: Boolean value True or False

Type of Testing: Unit Testing

Total Number of Test Inputs: 10

Test #1 Input Values: image_1_path , test_folder

Test #1 Expected Result: True, the image was deleted

Test #1 Result: True

...

Test #10 Input Values: image_1_path , image_1_path

Test #10 Expected Result: False, the image should not be deleted because of improper folder

Test #10 Result: True

In this scenario for Test #10, the expected result and the actual result are different. Meaning, there is a defect in our program and we need to modify our program to cover all test cases. Shown below is a table of the other functions we plan to perform unit tests on. This table includes a description of each unit, the boundary value, the input value, and the expected output.

Unit	Description	Boundary Value	Example Input	Expected Response
Classify model	Users can use our AI model to classify the specie of birds	A png or jpg file	Classify: bird.png	Our AI model will tell the species according to the database.
Save picture	User can save the picture in the backend	A png or jpg file	Save: bird.png	The picture that user saved will be stored in our database.
Delete pictures	User can delete the picture he doesn't like	A png or jpg file	Delete: bird.png	The picture will be deleted from our database.

As far as performing unit tests for the UI, these tests will look slightly different than the backend tests we perform. As mentioned previously, there are two styles of testing, black-box testing and white-box testing. For testing the UI, we will most likely be using black-box testing in which the expected result should be shown on the web application itself. An example of this might be structured in our test plan will be shown below.

Unit: upload image on user profile

Boundary Values: image with extension of png or jpg

Standard Output Value: image posted onto users images page

Type of Testing: Unit Testing

Total Number of Test Inputs: 10

Test #1 Input Values: bird.png

Test #1 Expected Result: image posted onto user image page

Test #1 Result: image posted onto user image page

...

Test #10 Input Values: myfile.txt

Test #10 Expected Result: flash message to user stating to select something with ".png or .jpg" extension

Test #10 Result: flash message to user stating to select something with ".png or .jpg" extension

In this scenario for Test #10, the expected result and the actual result are the same. Meaning, the test case we have conducted is accurate and there is no need to modify

our code for a situation like this. Shown below is a table of the additional units for our UI, a description of each unit, the boundary value, input value, and expected output.

Unit	Description	Boundary Value	Example Input	Expected Response
Login account	Users can login in their existing account or register an account	An email An password	Email: cl2445@nau.edu Password: 123456	If the email cannot match the password, the user will be rejected. If a user input the right password in the database, the user can log in successfully.
Make a post	User can make a post in the homepage	A post with text and picture	Text: Hello Picture: bird.png	User will have a post in the homepage and other users can see it, if the user attach a wrong file, he will be reminded.
Upload pictures	User can upload a picture into his gallery	A png or jpg file	Picture: bird.png	User will see the files he uploaded in the gallery, and if the user attach a wrong file, he will be reminded.
Comment	User can comment on other user's post	A comment of text	Comment: Hello, I like your post.	User can view his comment in the homepage

After performing a series of unit tests on the logical parts of our program and the user interface, we can now explain how we plan to implement integration tests.

Integration Testing

Integration testing is another incredibly important method for testing software, and one that is critical for all non-trivial applications and software packages. In the case of a web application, like pEEp, unit testing is used to show that small units of code produce correct results, whereas integration testing focuses on ensuring that the exchange of results between separate units is done correctly. Because we have implemented the MVC (model-view-controller) design pattern, our task for integration testing is to ensure that all communication between modules relays correct results, eliminating any possibility of undefined behavior. Specifically, we must ensure that the following interactions always produce consistent results and behavior:

1. Database queries to the model from the controller
2. Injection of data into view templates by the model

1. Database queries to the model from the controller

Database queries are done only in one place: the routing functions that handle all possible requests. To ensure that these queries return what we expect, we can create a test database with relatively few entries and add some logging statements to the routing functions that conduct queries. These logging statements will include information that allows us to see what the query and result is so that we can compare the expected and actual results.

As an example, let's choose a simple route that conducts a query, and modify it so that we can see how logging will work. Let's choose the following route for displaying a single post:

```
@posts.route('/post/<int:post_id>')
def post(post_id):
    post = Post.query.get_or_404(post_id)
    return render_template('post.html', title=post.title, post=post)
```

First of all, we will need to set some variables. Specifically, we will need a file path and file name that we can use to open a file for writing. Second, we will want a global logging flag that will toggle the logging functionality on and off. We can simply put both of those at the top of the `peep/posts/routes.py` file for now:

```
logfile_path = "peep/log/posts.lgf"
LOGGING = True
```

With this information, we can implement our simple logging logic. The newly transformed post route will now log to the log file specified if the global flag is set:


```

@posts.route('/post/<int:post_id>')
def post(post_id):
    post = Post.query.get_or_404(post_id)
    if LOGGING:
        # Writing to file
        with open(logfile_path, "w") as file:
            file.write(f"author: {post.author}, date_posted: {post.date_posted}\n \
                content: {post.content}\n")
    return render_template('post.html', title=post.title, post=post)

```

Because the test database has relatively few entries, it will be easy to confirm that the query gives us what we expect through a simple file comparison. This same method can be used on all other routes, and in this way we can verify that all queries conducted by the controller consistently produce the correct results.

2. Injection of data into view templates by the model

As mentioned in previous documents, Flask (our backend framework) uses a templating engine called Jinja2 to render dynamic templates. This templating engine allows us to write real code inside of our html templates. To continue with the example of a single post, let's look at the templating code to see how this is done dynamically (some code has been removed for demonstration):

```

{% extends "layout.html" %}
{% block content %}
    <article class="media content-section">
        
        <div class="media-body">
            <div class="article-metadata">
                <a class="mr-2"
                    href="{{ url_for('users.user_posts', username=post.author.username) }}">
                    {{ post.author.username }}
                </a>
                <small class="text-muted">{{ post.date_posted.strftime('%Y-%m-%d') }}</small>
            </div>
            <h2 class="article-title">{{ post.title }}</h2>
            <p class="article-content">{{ post.content }}</p>
        </div>
    </article>
{% endblock content %}

```

As we can see, the **post** variable that is given to this template during the call to `render_template` in the post route is the same post variable that we received as a result of querying the database. By passing this post variable to the template, Jinja2 is able to dynamically resolve all code with

the content that it refers to. For example, `{{ post.content }}` will be resolved by injecting all data from the content field of the post into the html paragraph that the code is inside of.

To verify that this is done correctly, all we must do is create a new post, and then navigate to the page where this post is displayed. If any references cannot be resolved, then an error message will be raised on the server, and a HTTP code of 500 will be sent back to the user with a message saying "Something went wrong". In this way, it takes nothing but a visual inspection of the post page and its content to verify that data is being correctly passed to the templating engine and properly injected into the html pages

These two interactions comprise the primary module integrations that must be tested. It should be noted, however, that the AI module is not a distinct module from the controller, so it was not included in integration testing. Because the AI model can be used to perform inferences by simply loading the model file and the specified framework, it is trivial to show that it integrates into the controller (it's parent module, if you will).

Using the method described in both of the above examples, we can test all database queries to examine the exact queries taking place, the results from said queries, and compare the actual and expected results. Likewise, for the view and controller integration, all we need to do is run the application and ensure that all templating code references are resolved properly. Since the database queries have already been examined for correctness, we can be sure that the information we are accessing from within the templates will resolve properly. After abstracting the tests in these examples to the entire codebase, we can be sure that our three primary modules integrate correctly and consistently if they pass all of our explicit integration tests.

Usability Testing

Usability testing is a technique that is focused on the user's interaction with our system, and what functionality the user interface offers to the users. When examining the “usability” of our system it is important we take note of the following three points. Is our system 1) effective, 2) engaging, and 3) easy to learn. These will be our three major points of testing. Each one will have unique testing criteria that will tell us if the aspect has been achieved within our system. As of now, our pool for testing will consist of only one user. This user is our client, David Plemmons. We will ask him to interact with our web application and then later record the results in a design review. Since this web application is being built for him, and to his standards, it is imperative that we meet our usability aspects and provide him with the best experience possible and take all of his feedback seriously.

Effective:

Definition: Successful in producing a desired or intended result.

Goal: From day one our client has made it known that he wants all the focus to be towards the birds. For our web application to be considered effective it is important that the birds are the center of attention. The other elements should have a dull but professional look, to make the images stand out.

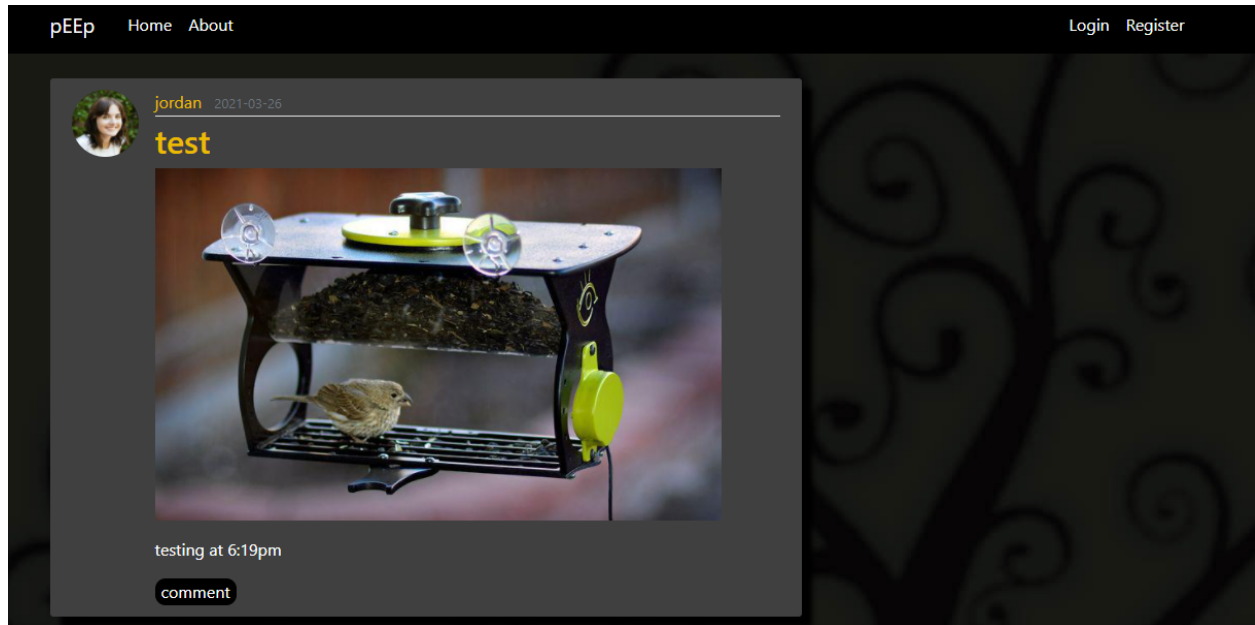
Test Criteria:

User Questions:

1. When viewing the homepage what is the first thing the user sees?
2. When navigating through the web application, is there anything that draws the users attention away from the bird images?
3. Does the web application appear official or professional looking?

User Tasks:

1. Open website.
2. Use the navigation bar to navigate through different pages.



Our System: The above screenshot is of our current rendition of the home page. Notice how the background, text, post panel, and navigation bar are all dull and mute colors. This is done purposefully to draw your attention to the bright and colorful photo of the bird. With this layout regardless of where you are looking on the page the center point will always be the bright and beautiful pictures posted by the user.

Engaging:

Definition: Tending to draw favorable attention or interest.

Goal: Having an engaging web application is key. We are not building a web application for the DMV, we want to build something that can be considered as “entertainment” for our users. Something that a user will be inclined to enjoy during their free time. It is important that the user experience does not feel like a chore but rather something they would want to do instead of chores.

Test Criteria:

User Questions:

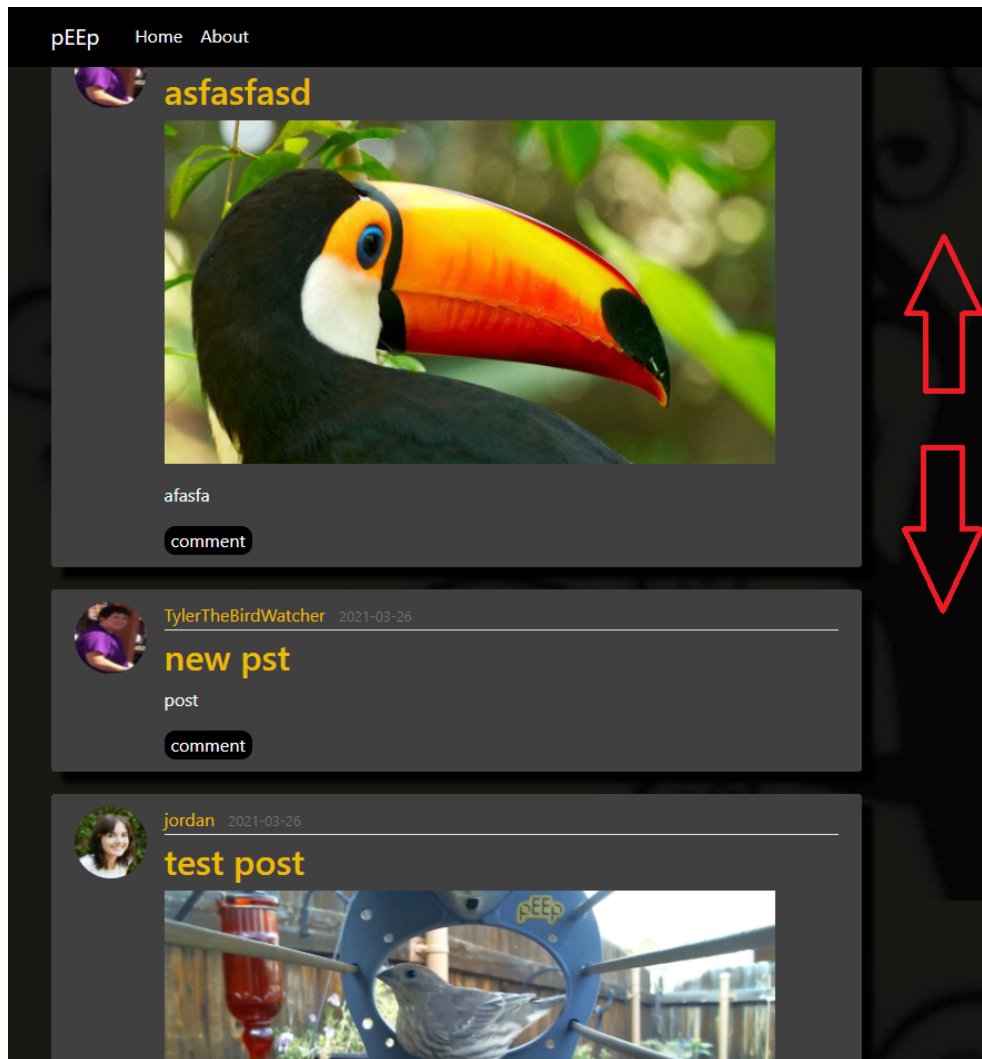
1. Is the user experience enjoyable?

2. Is there anything the user would change about interacting with the web application?
3. Is everything readable?
4. Does the user feel any major features are being left out?

User Tasks:

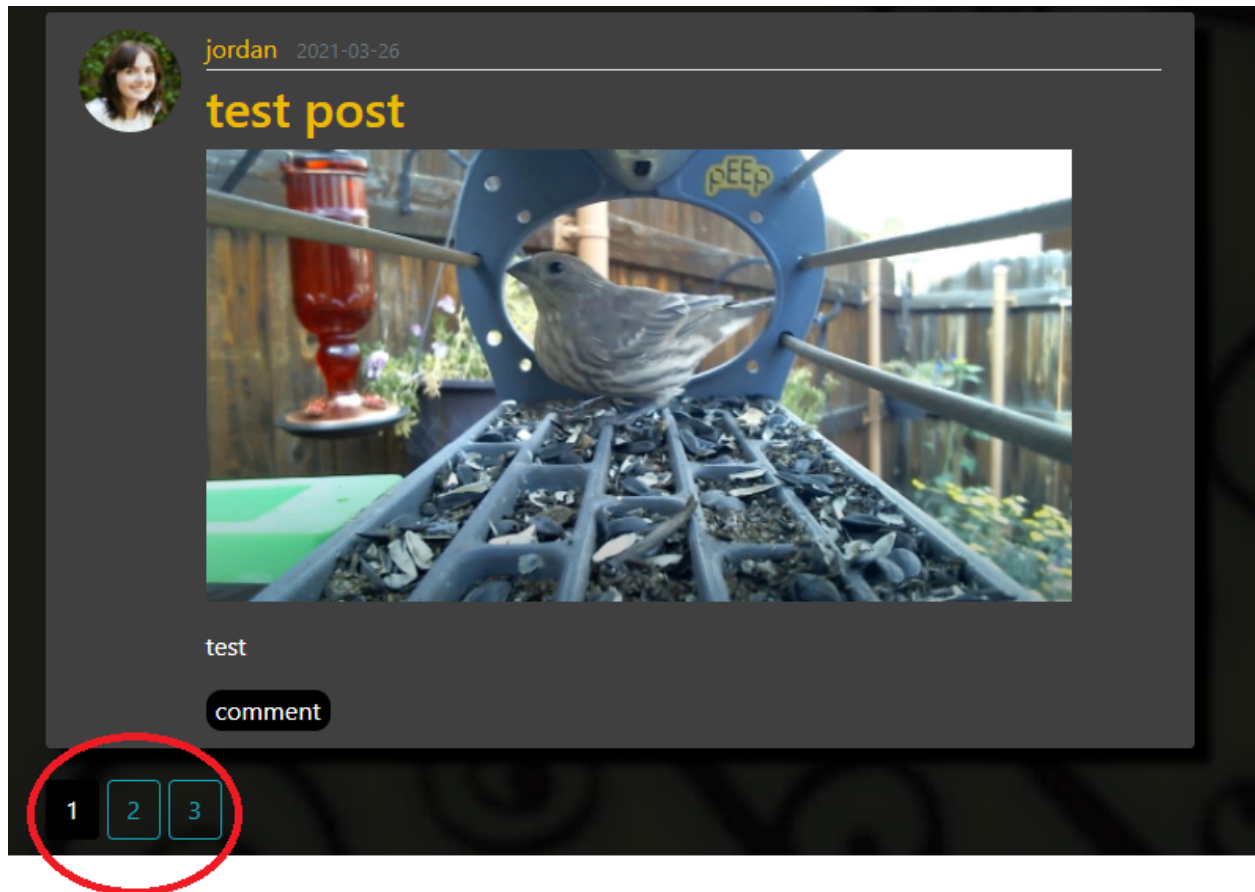
1. From the homepage view other users posts.

Our System:



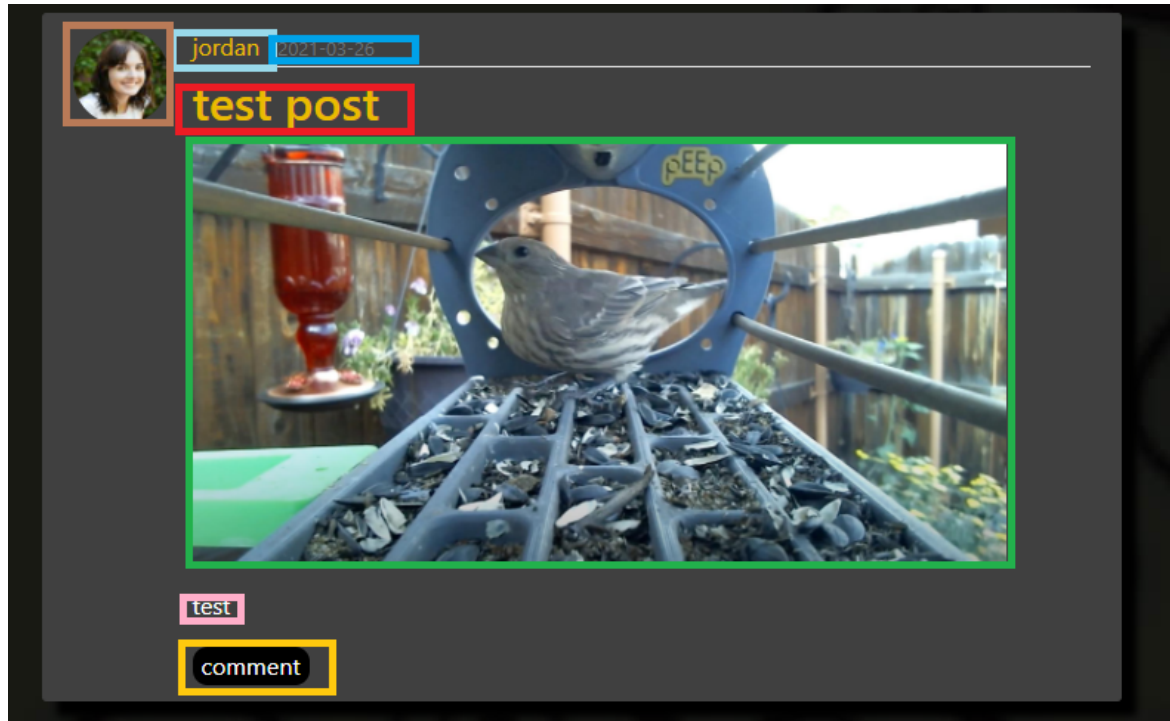
This is another screenshot of our homepage. As mentioned previously the main attraction of our web application is the birds. This is why it was imperative for us to make viewing of said birds simple but efficient. Scrolling down the homepage you will be able to view all posts on the available page. Scrolling back up will allow you to view

previous posts. Once the bottom of the page has been reached, you are only one click away from a fresh new page of content. As seen by the screenshot below.



This is a simple but practical design that can allow users to casually browse the most recent bird watching community content. All posts are dated accordingly so the user will always know how recent the post has been made.

A post itself consists of seven main elements.



1. **Profile Picture:** an easy visual identifier as to who made the post. (represented brown box)
2. **Name of poster:** the username or title the individual who made the post goes by. (represented by the light blue box)
3. **The date posted:** note this is not the current date, but rather the date at which the post itself was made. (represented by the blue box)
4. **The post title:** the name given to the post by the user who posted it. (represented by the red box)
5. **The post picture:** The centerpiece of post. (represented by the green box)
6. **The post content:** Any additional text information that the user would like to add to the post. (represented by the pink box)
7. **Comments:** This section allows for other users to interact with a post by posting text based messages such as reactions or overall thoughts on the post itself. (not yet implemented) (represented by the yellow box)

This design allows for plenty of content and user interaction within each post while still maintaining focus on the pictures themselves.

Easy to Learn:

Definition: software that is well designed and capable of being used without having to wade through documentation.

Goal: It is understood that the audience of our web application is that of an older crowd and that they might not be the most technically savvy. Working with a community based system it is important that the start up time for a new user is streamlined and that existing users can dive right in without need of instruction.

Test Criteria:

User Questions:

1. Was any aspect of the web application trivial?
2. Was it easy to make an account?
3. Was it easy to recover the user's password?
4. Was it easy to Login?
5. Was it easy to make a Post?
6. Was it easy to upload an image?
7. Was it easy to delete an image?
8. Was it easy to identify an image?
9. Was it easy to see other users' posts?
10. Was the web application easy to navigate?
11. Is there anything the user would change about the layout to make things easier?

User Tasks:

1. Register an account.
2. Attempt to login with the wrong password.
3. Select forgot password and check email to recover password.
4. Change password.
5. Login with email and new password.
6. Select "Account" from the top navigation bar.
7. Upload a new profile picture and change email if preferred.
8. Select "Upload images" from "Actions" navigation bar.
9. Upload an image from a local device.

10. Select “My Images” from “Actions” navigation bar.
11. Favorite the Image.
12. Identify the image.
13. Delete the image.
14. Select “Create a Post” from “Actions” navigation bar.
15. Create a post with an uploaded image.
16. Select “My Posts” from “Actions” navigation bar and check if the post is there.
17. Select “Home” from the top navigation bar.
18. Check that post has been made publically.
19. Select “Logout” from the top navigation bar.
20. Refresh page to ensure that the user is logged out.

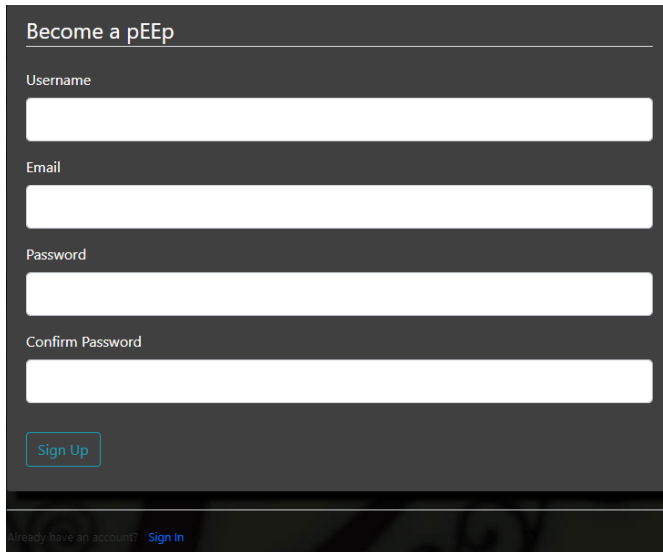
Our System:

Registering an Account:

From the homescreen, at the top of the page users will find a navigation bar. All the way to the right you are given the option to register. This will allow new users to create an account and begin their pEEp experience.



Once there they will be brought to a page where they can fill out their personal user information.



The image shows a registration form titled "Become a pEEp". It features four input fields: "Username", "Email", "Password", and "Confirm Password". Below the "Confirm Password" field is a "Sign Up" button. At the bottom left, there is a link that says "Already have an account? Sign In".

Once you have selected a username, password, and entered their desired email address they will be able to click sign up. Note that if a user forgets their password it can always be reset.

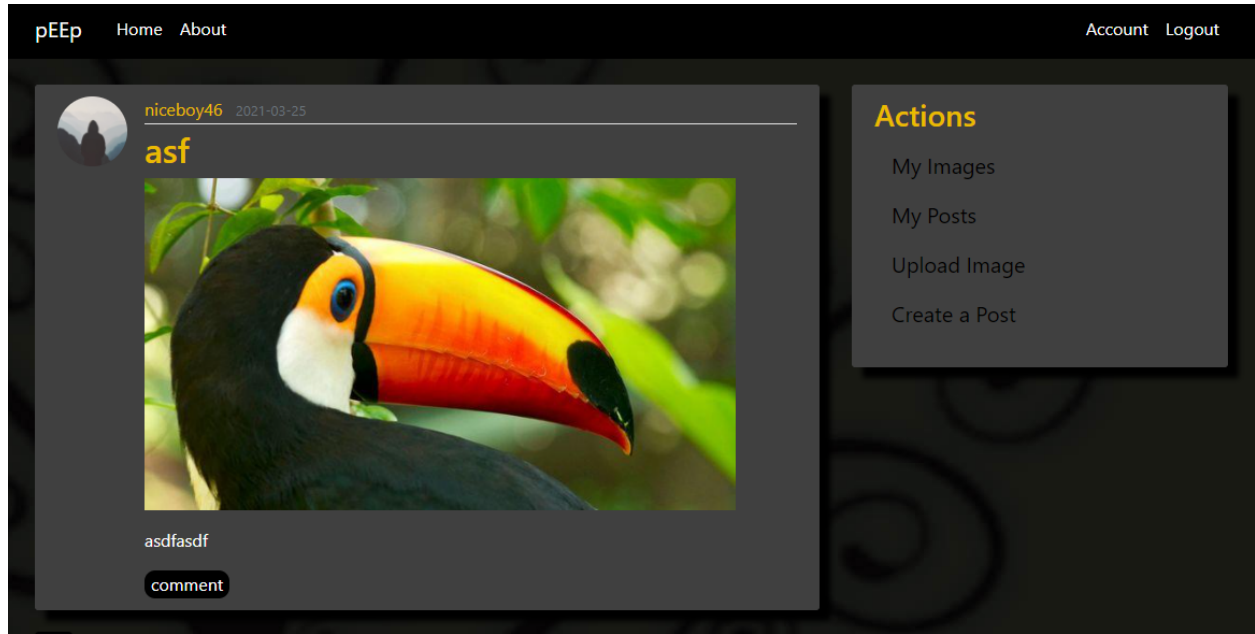
Existing User Login:

If a user already has an account with pEEp they simply need to click Login from the home page and enter their login information consisting of their email and password.



After Logging In:

Once logged in users will notice a new set of navigation tools under the "Actions" sidebar on the right side of the page.

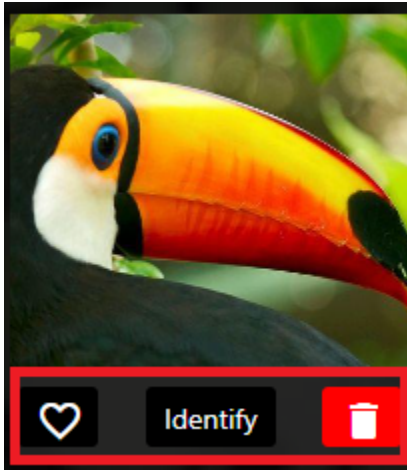



Users are now given four options for interacting with pEEp.

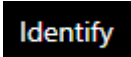
1. My Images - will allow users to view all images they have uploaded to their pEEp library. Note these images are simply bound to your account and have not been posted yet.




Each image has three interaction icons below it:

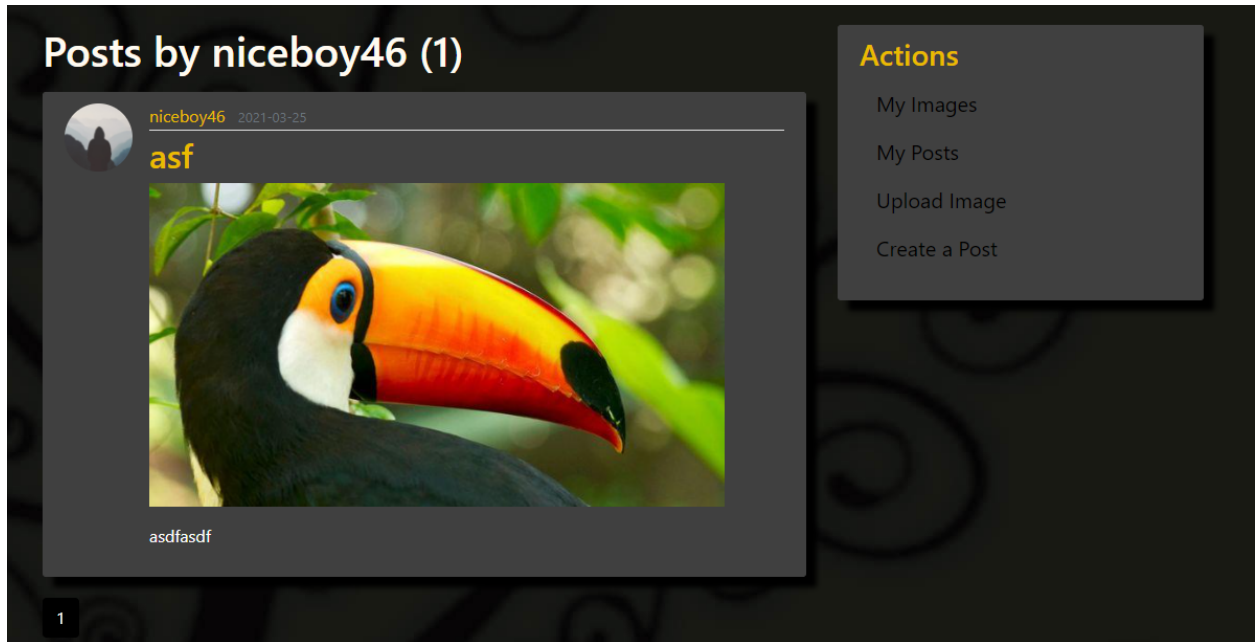


The  icon will allow users to favorite the image.

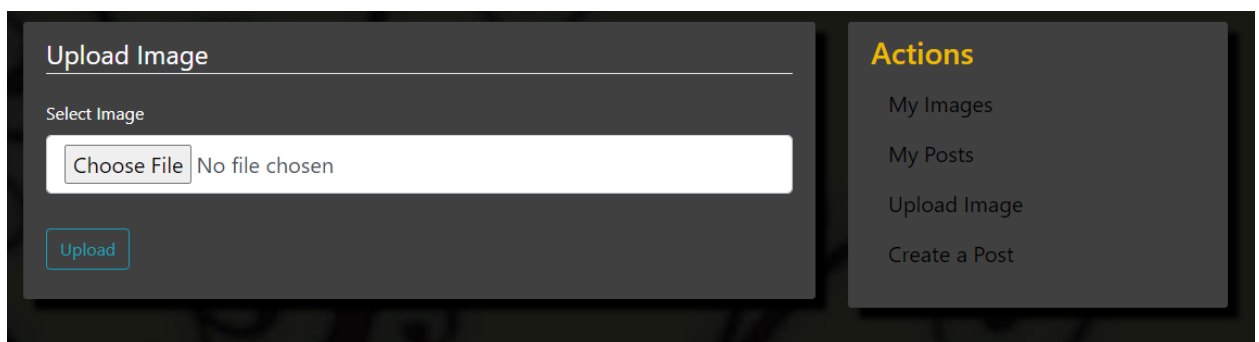
The  icon will allow users to run the image through our AI model to see what type of bird they have captured.

The  icon allows users to delete the image.

2. My Posts - allows users to view all posts that they have made in similar fashion to the “My Images” section.

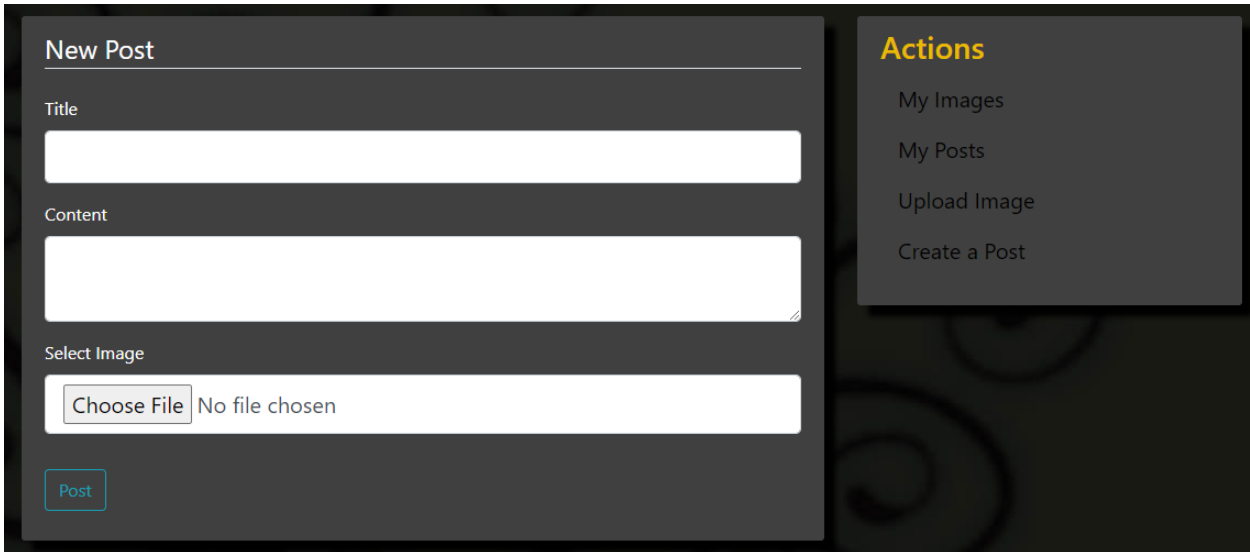


3. Upload Image - a simple interface that allows users to upload images. Note these images are not posted, they are uploaded to your “My Images” page.



Users can upload images in two steps:

1. Click “Choose File” and select the image you wish to upload from your device.
 2. Click upload.
4. Create A Post - interface that allows users to create a post for all of the pEEp community to see!



The image shows a dark-themed user interface for creating a new post. On the left, there is a 'New Post' form with three main sections: 'Title' with a text input field, 'Content' with a larger text area, and 'Select Image' with a 'Choose File' button and the text 'No file chosen'. Below these is a 'Post' button. On the right, there is an 'Actions' menu with a yellow header and four items: 'My Images', 'My Posts', 'Upload Image', and 'Create a Post'.

Users can create a post in four steps:

1. Choose a title.
2. Type any additional information regarding the post in the content section.
3. Select an image from their local device by clicking Choose File.
4. Click Post

Once a user has performed the desired actions they can then return to the homescreen by clicking home in the top navigation bar and continue browsing the pEEp community.



We have designed the web application so that nothing is trivial or left to question. No two pages look the same so there is no debate on whether you are uploading an image to your “My Images” page or if you are posting the image. With the “Actions” navigation window only appearing when users are logged in they will always know if they are logged in or not. The design is simple yet inviting and caters perfect to that of a less tech-savvy audience. Any new user should be set up and posted in a matter of minutes. Since usability testing is focused on the interactions between the software system and the end user, and is intended to ensure that users can effectively access the functionality provided. This type of testing examines the overall quality and understandability of the user interface exposed by your system and the workflow that your system embodies; it is vitally important for end-user facing applications, especially when users are not particularly patient or technically savvy.

Conclusion

In this software testing plan, we detail the plan for performing software tests and provide in-depth examples of the tests to be performed. As a team, we broke down the main functionality of the web application to develop tests to ensure that each unit of code performed as expected and remained reliable. We have planned three types of tests: unit tests, integration tests, and usability tests. Each type of test has the potential to discover unexpected behavior or errors that other types of tests cannot easily detect, ensuring that we cover all bases and verify the correctness of all code.

Unit tests ensure that each piece of code works correctly, which helps to find errors in individual components. This is done by providing units of code with both correct and incorrect input, being explicit about what the expected output is for each input, and then comparing the expected and actual results to make sure that they align. Integration testing ensures that components are working together and communicating correctly, which is done by checking components that need to communicate with each other. Specifically, for the pEEp web application, integration testing is conducted to ensure that the model, view, and controller modules all communicate with each other in a consistent and correct manner.

Finally, through our usability testing, we will allow our end users to provide feedback on various aspects of the application that need to be improved so that we can optimize the user experience. In our case, we will have our client, David Plemmons, run through a sequence of tests that we have designed, recording his responses as he goes. Using the results of these three tests, we will then make changes to improve the software quality and user experience. Through this rigorous testing, we will ensure that our software functions correctly and, more importantly, that our web application is as seamless and easy-to-use as possible, allowing bird enthusiasts to flock to pEEp (pun intended) as the community grows.