# Wildlife Disease Website Redesign

By: James Kim, Maanas Muddam, Bryant Nguyen, Kristopher Moratti

Final Report

Client: Professor Luis Escobar

CS 4624, Professor Edward Fox

December 15, 2021

Virginia Tech

Blacksburg, VA 24061 USA

# Table of Contents

**Table of Figures**…………………………………………………………………….....4

**Executive Summary**…………………………………………………………………......7

**Introduction**……………………………………………………………………………......9

**Design Requirements**…………………………………………………………………...10

**Quality Analysis Testing**……………………………………………………………….11

**User Manual**………………………………………………………………………………12

Navigation Bar……………………………………………………………………………12

Footer…………………………………………………………………………………….....14

Landing Page……………………………………………………………………………....15

About Us Page…………………………………………………………………………....16

Publications Page………………………………………………………………………..18

Research Page…………………………………………………………………………....19

Contact Page……………………………………………………………………………..21

Administrator Login Page……………………………………………………………….23

**Developer's Manual**……………………………………………………………………24

Application Layer…………………………………………………………………….....24

Pages Layer……………………………………………………………………………..25

Static Data Layer……………………………………………………………………......26

Reusable Components Layer………………………………………………………....28

About Us Page Description…………………………………………………………...29

Contact Us Page Description…………………………………………………….......30

Home Page Description……………………………………………………………......31

Login Page Description………………………………………………………………...32

News Page Description………………………………………………………………...32

Publications Page Description…………………………………………………………33

Research Page Description…………………………………………………………...34

About Data Component……………………………………………………………...34

Button Component……………………………………………………………...38

Contact Page Component……………………………………………………….38

Footer Component……………………………………………………………....43

InfoBanner Component………………………………………………………....45

Login Component……………………………………………………………….....46

Navbar Component……………………………………………………………..47

Sidebar Component……………………………………………………………..48

Publications Component………………………………………………………..49

Research Component…………………………………………………………...52

Server-Side Interaction…………………………………………………………58

**Server Startup**………..…………………………………………………………..58

Data Models...………..……………………………………………….……...59

Endpoints…...………..…………………………………………………………63

**Lessons Learned**…………………………………………………………………75

Timeline………………………………………………………………………...75

**Deployment**………………………………………………………………………76

**References and Technologies**…………………………………………………77

Technologies…………………………………………………………………..77

References……………………………………………………………………77

**Acknowledgments**……………………………………………………………...78

# Table of Figures

Figure 1: New Landing Page Design -- Header…………………………………………….12

Figure 2: New About Us Page Design……………………………………………......13

Figure 3: Login Page Design…………………………………………………………….14

Figure 4: New Landing Page Design -- Footer……………………………………………..15

Figure 5: New Landing Page Design…………………………………………………….16

Figure 6: New About Us Page Design -- Cards…………………………………………….17

Figure 7: About Us Page – Expanded Card……………………………………………......17

Figure 8: Add Card – About Us Page……………………………………………………18

Figure 9: New Publication Page Design…………………………………………………19

Figure 10: New Research Page Design…………………………………………………….20

Figure 11: Research Page Card Expanded………………………………………………….20

Figure 12: New Contact Us Page Design………………………………………………….21

Figure 13: Contact Us Page Form……………………………………………………….22

Figure 14: Filled Out Contact Page Form………………………………………………....22

Figure 15: Sample Email from Contact Form……………………………………………..23

Figure 16: Administrator Login Page……………………………………………………...23

Figure 17: App.js File Location……………………………………………………….....24

Figure 18: App.js Router Calls…………………………………………………………....25

Figure 19: Pages Directory Location……………………………………………………...26

Figure 20: Function for Displaying Home Page……………………………………………..26

Figure 21: Static Data Directory Location………………………………………………...27

Figure 22: Component’s Directory Location……………………………………………...28

Figure 23: AboutData Directory Location………………………………………………...28

Figure 24: AboutPageDisplayer.js File……………………………………………….....30

Figure 25: ContactPage.js File……………………………………………………….31

Figure 26: HomePage.js File……………………………………………………….......31

Figure 27: Login.js File……………………………………………………………...32

Figure 28: NewsPage.js File……………………………………………………….33

Figure 29: PublicationsPage.js file……………………………………………………….33

Figure 30: ResearchPage.js File……………………………………………………….34

Figure 31: Button Component Implementation.…………………………….…………39

Figure 32: EmailJS Available Service……………………………………………...40

Figure 33: EmailJS Available Service……………………………………………...40

Figure 34: Location of Contact.js……………………………………………………..41

Figure 35: ContactPage.js – Form Definition……………………………………....42

Figure 36: ContactPage.js – Styling Elements……………………………………...43

Figure 37:  Sample Template………………………………………………………43

Figure 38: Form Input Field Example……………………………………………...43

Figure 39: Email Template………………………………………………………....44

Figure 40: Form Submit Function -- Contact.js…………………………………....44

Figure 41: Footer.js File…………………………………………………………....45

Figure 42: FooterElements.js Example Styled Constant………………………………..46

Figure 43: FooterFunction.js………………………………………………………46

Figure 44: handleFormSubmit() function…………………………………………47

Figure 45: login() Function……………………………………………………..48

Figure 46: Navbar.js Function…………………………………………………..49

Figure 47: PublicationPost.js Function…………………………………………....51

Figure 48: Get and filter publications functions……………………………………..52

Figure 49: PublicationsPosts component……………………………………………53

Figure 50: Categories Component of Research……………………………………...55

Figure 51: ResearchPosts Component…………………………………………......56

Figure 52: Research Component……………………………………………………57

Figure 53: ResearchPost Component………………………………………………..58

Figure 54: Server.js Server Startup...……………………………………………….60

Figure 55: User Data Model………..…………………………………….……….61

Figure 56: AboutPost Data Model………..…………………..……..…………………62

Figure 57: ResearchPost Data Model………....………………..……..…….……..……63

Figure 58: PublicationPost Data Model………....………………..……...….……...64

Figure 59: Successful POST Request to Auth Endpoint……….......…….………..……65

Figure 60: Authenticated Endpoints Require a JSON Web Token…....….………..…...66

Figure 61: Retrieve all ResearchPost Instances……….….….......…….………..…..67

Figure 62: Successful ResearchPost Creation..………...……….......…….………..…..68

Figure 63: Successful ResearchPost Removal………...……….........…….………..…69

Figure 64: Retrieve all PublicationPost Instances………...………....…….…………...69

Figure 65: Successful PublicationPost Creation..………...……….......…….….…...…70

Figure 66: Successful PublicationPost Removal………...……….........…….….……..72

Figure 67: Retrieve all AboutPost Instances………...……….......………...………….73

Figure 68: Successful AboutPost Creation..………...……….......……...….……………74

Figure 69: Successful AboutPost Removal………...……….........……...….……………75

Figure 70: Timeline Chart………………………………..……………..………..76

## Executive Summary

Our client, Professor Luis Escobar, has a website located at https://ecoguate2003.wixsite.com/escobar, which is used to share information and news about the laboratory. This website is currently being maintained by Professor Escobar and his students. As a result, they lack the necessary skills to develop and maintain a site that has the desired functionality needed to represent the laboratory.

Our task was to redesign the website in a way that meets certain design and functionality expectations in order to provide information about the laboratory. Those design expectations were as follows:

- Minimalistic Design
- Informative and Simple User Interface

All other design aspects were left up to the team and were subsequently approved by Professor Escobar. As for the functionality, Professor Escobar requested the ability to share relevant information in the following formats:

- News
- Publications
- Information about Research
- Information about Lab Members

In addition to these design aspects, the website needed to be created in a way that is easily documented and hosted on a domain that is free of charge. All of this is done to achieve the sustainability of the website.

This led to a front and back-end design strategy following the MERN stack paradigm. The MERN stack is built based on the following technologies:

- MongoDB – Document Database
- Express(.js) – Node.js Web Framework
- React(.js) – Client-side JavaScript framework
- Node(.js) – The premier JavaScript web server

Utilizing this paradigm, the team created a website whose design outline reflected the organization of the original website. That is, we had the following pages:

- Home Page
- Publications Page
- About us Page
- Research Page
- Contact Us Page

These pages contain the same or similar information to the original website; however, the functionality and organization has been changed. Overall, the task was to redesign a website that meets modern design standards and functionality while being sustainable.

# Introduction

This Final Report is designed to provide a guide regarding the creation and use of the updated website for Professor Escobar's Laboratory located at wildlife.cs.vt.edu. As a whole, the report provides a summary of the design requirements, the means of realizing those requirements, a user manual, and a developer's manual.

## Design Requirements

The primary front-end design requirements were as follows:

- Minimalistic Design
- Informative and Simplistic User Interface

These requirements were to enable users to easily navigate the website while providing a desirable aesthetic. This required moving all of the original data from the old website to the new website and refactoring the data to fit within the new design paradigm.

Data was taken from the original website in the form of lab bios, publication information, research posts, and logos. This required manually entering in the required information into static data structures within the new website. The most numerous entries were taken from lab bios followed by publications followed by research posts. After manually acquiring the data, the data was rendered in the format shown in the User Manual.

## Quality Analysis Testing

The Quality Analysis was done according to the following steps:

- Validation Testing
- Browser and Device Testing
- Input Testing
- Performance and Bug Testing
- Accessibility Testing

Validation Testing consisted of ensuring the product properly meets the design specifications of the client as well as the design standards set by the industry. The first step is ensuring proper HTML markup by running the relevant code through a markup validator such as W3C Markup Validation Service [5]. This ensures that website design complies with web standards. The second step in validation testing is ensuring all media content is functioning properly and that the text displayed on the site is proper and up to date. This includes checking the endpoints for hyperlinks as well as checking media, such as images, for proper content display.

Browser and Device Testing consisted of testing the website display and functionality on multiple browsers, devices, and screen sizes to ensure capability for different types of users. To accomplish this, the website was viewed across multiple browsers as well as mobile formats and the content display and functionality were evaluated to be satisfactory across devices.

Input Testing consisted of testing the form and input fields for the website. Primarily, it involved ensuring relevant fields could handle a wide range of alphanumeric combinations and that these fields were properly displayed in the cards. Also, text fields were trimmed according to the current view of the page, that is, expanded card values had different text lengths than cards in their normal format.

Performance and Bug Testing consisted of testing the front-end and back-end interaction. This involved validating the data sent from the website to the database and back to the website to ensure the right content is sent and displayed for the user. Furthermore, all clickable text and buttons are tested to ensure they are performing their desired task.

The final part of the team’s internal QA testing was Accessibility Testing. This involved a final verification of the site’s content display for all users including those with impairments of any kind. To the best of the team’s ability, suitable colors, resolution display, and alternative forms of content were provided to ensure universal access for users.

# User Manual

The following provides a detailed look into the possible user interactions on each page. Within each page, there will be a description of its contents followed by a description of the functionality provided within.

## Navigation Bar

The first major component of the website is the navigation bar highlighted in red in Figure 1.

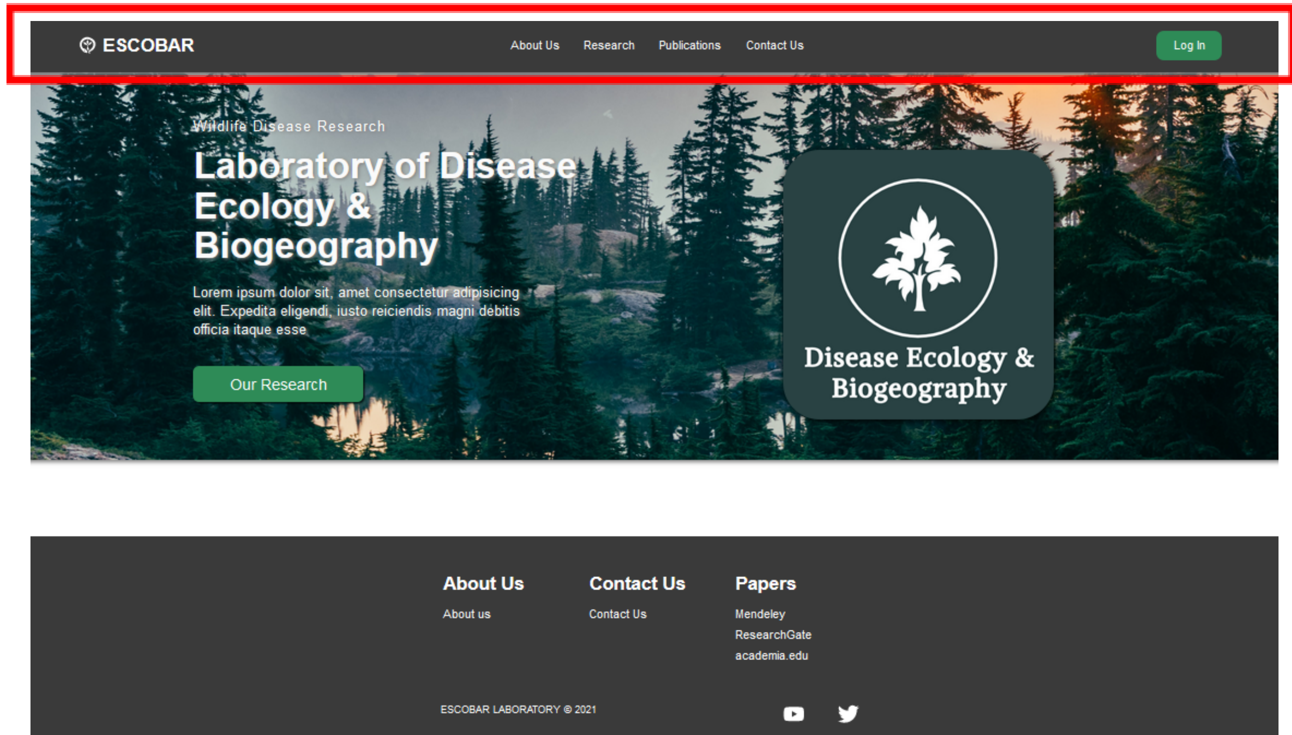



Figure 1: New Landing Page Design -- Header

There are 5 pieces of clickable text in the navigation bar seen in Figure 1 providing similar functionality. Those pieces of text are the logo in the top left, “About Us”, “Research”, “Publications”, and “Contact Us”. When clicked, the user is navigated to the page representing the clicked text. That is, clicking the “About Us” text brings the user to the About Us page shown in Figure 2.

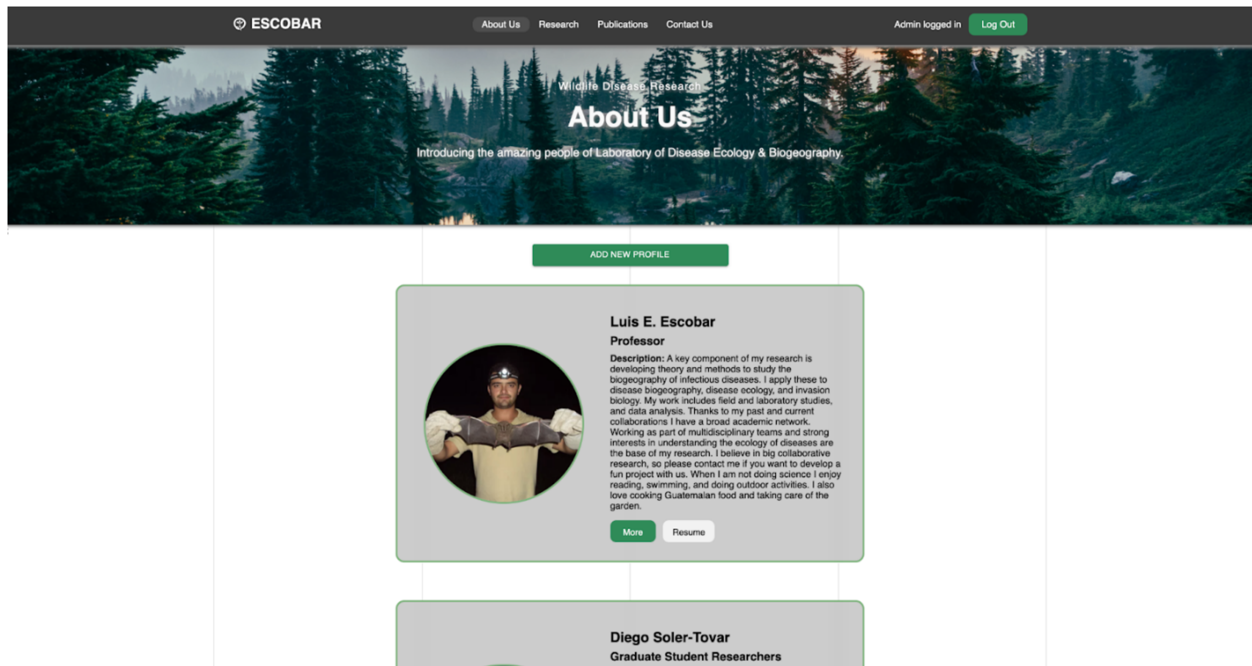



Figure 2: New About Us Page Design

The “Escobar” logo in the top left also provides clickable navigation. When clicked, the user is redirected to the home page seen in Figure 5.

Located in the far right of the navigation bar, there is a green “Log In” button. When clicked, the user is redirected to a login page (see Figure 3) for administrators.

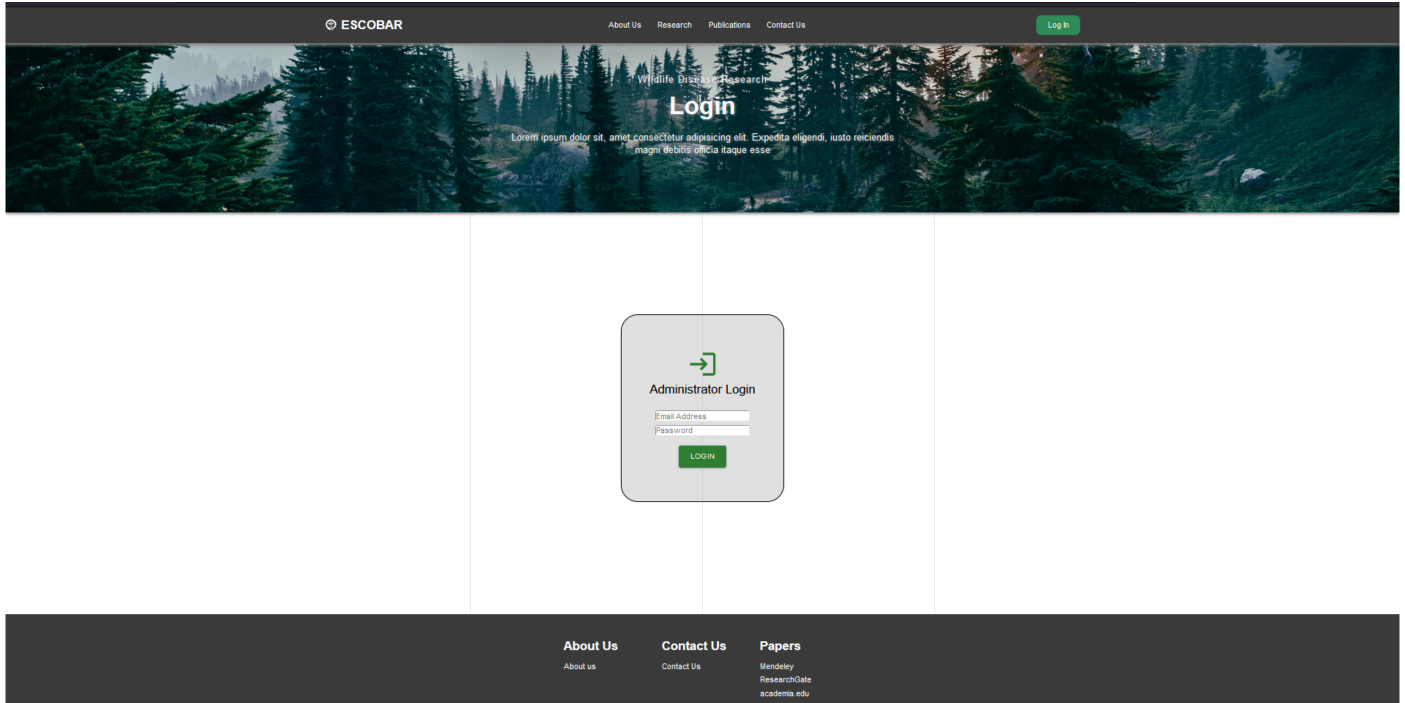
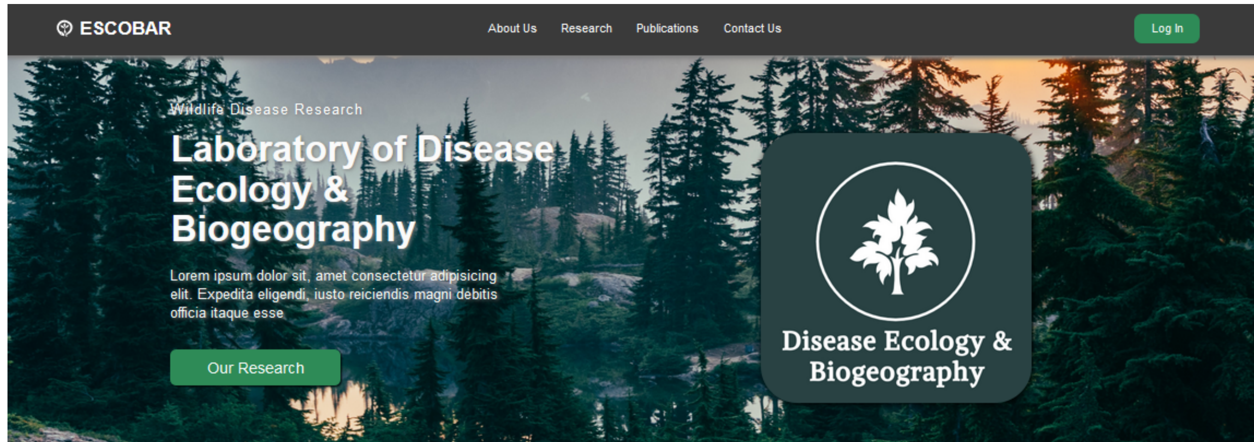



Figure 3: Login Page Design

Professor Escobar is given the administrator login information and if entered into the text fields correctly, clicking the Login button will authenticate the user and provide edit features for the site.

## Footer

The second major component of each page is the footer contained in the red rectangle shown in Figure 4.



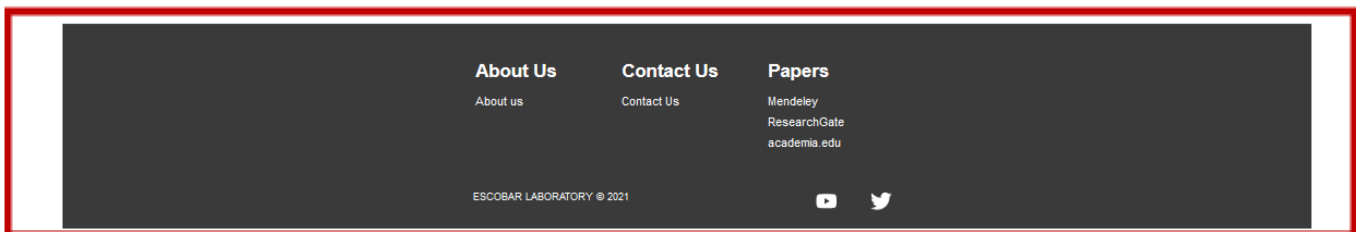




Figure 4: New Landing Page Design -- Footer

There are three sections: “About Us”, “Contact Us”, and “Papers”. The first two sections contain clickable text that navigates the user to the page described by the text, i.e., clicking the “Contact Us” text underneath that section header navigates the user to the Contact Page shown in Figure 12. The third section titled “Papers" provides clickable text that navigates the user to a new web page containing the works published by the laboratory at that link. For example, clicking “Research Gate” brings the user to https://www.researchgate.net/profile/Luis-E-Escobar.

Finally, there are the Twitter and YouTube logos located at the bottom of the footer. Both these logos are clickable and navigate the user to the following links: https://www.youtube.com/channel/UCOkukMnDkFmdcusq8CjCvLQ/feed (Luis Escobar’s YouTube Page) and https://twitter.com/Luis_Escobar_VT (Luis Escobar’s Twitter Page).

## Landing Page

Shown in Figure 5 is the currently designed home or landing page.

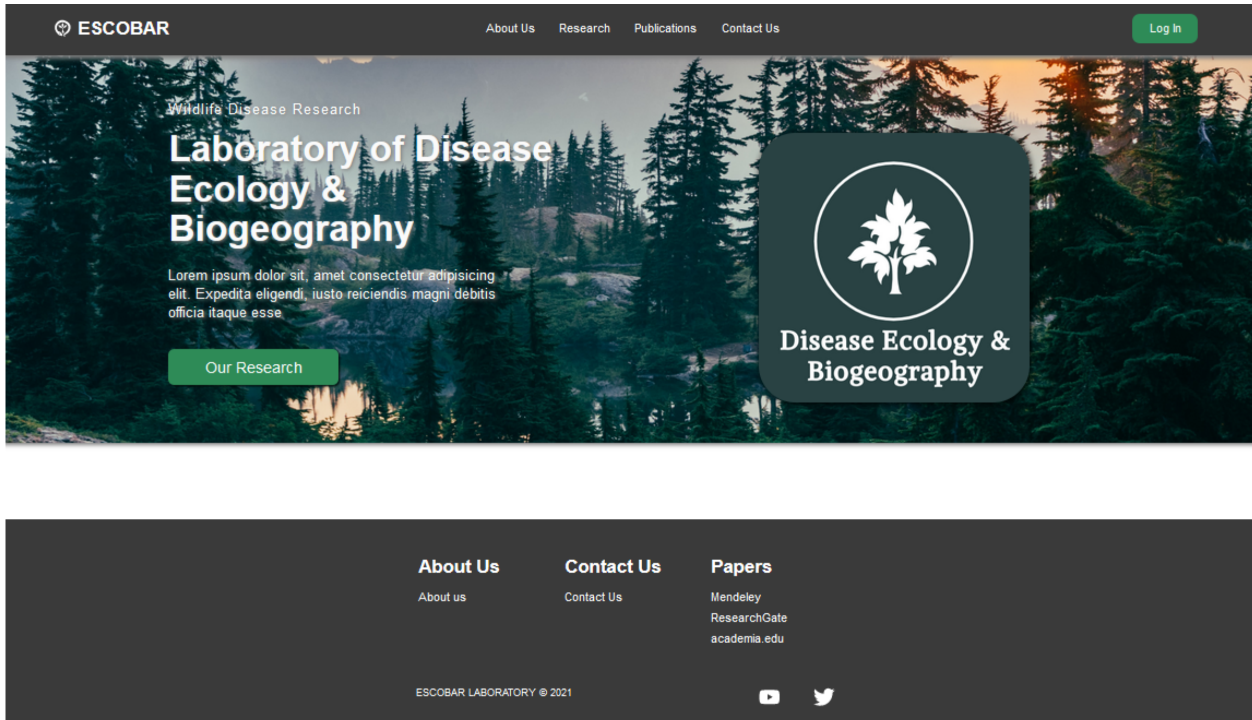



Figure 5: New Landing Page Design

The landing page purpose is to provide the user with an introduction to the laboratory and a start point from which the user can traverse the pages. This page contains a few notable components for the user. At the top of the page is a header with the primary purpose of providing the user with navigation (see Navigation Bar Section). At the bottom of the page is a footer whose purpose and features are described in the "Footer" section.

In the middle of the page, there is a brief introduction to the laboratory followed by a green "Our Research" button. This button is clickable. When clicked, the user moves to the Research page seen in Figure 10.

## About Us Page

The About Us page provides the user with information about the laboratory's professors and students. Each member of the laboratory has an associated card highlighted in red seen in Figure 6.

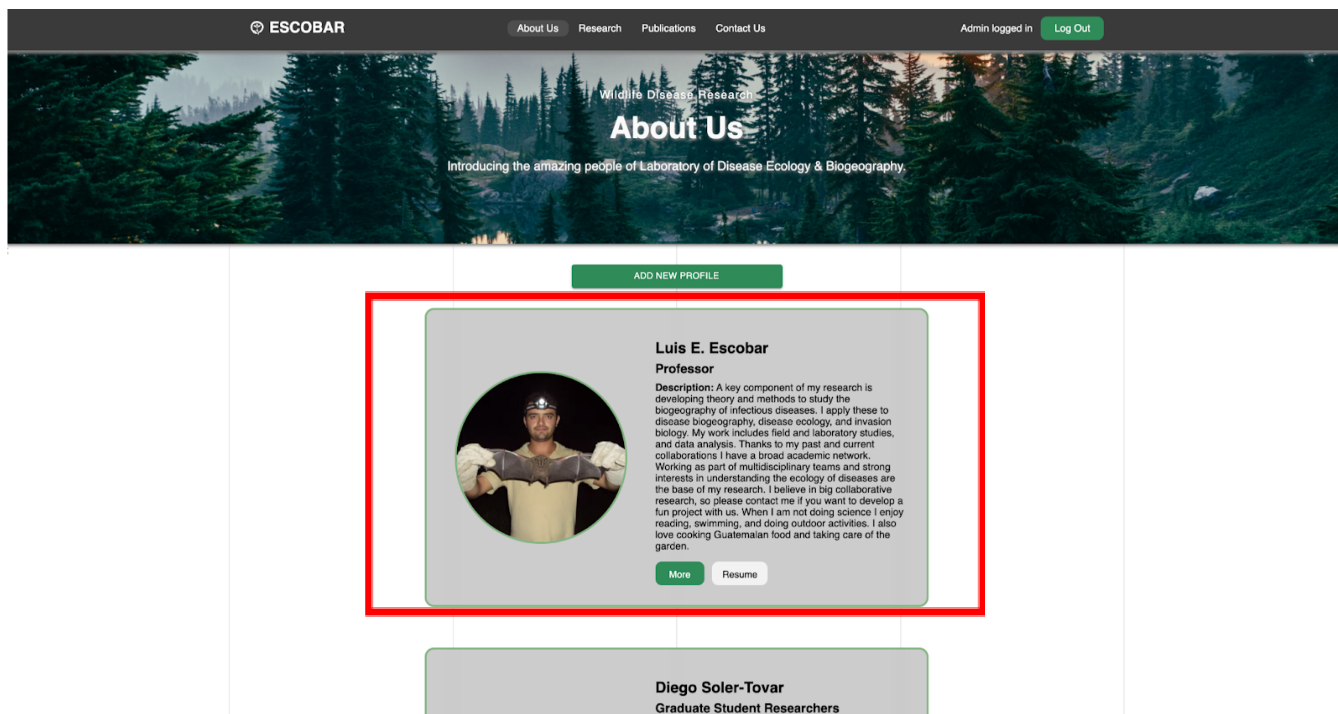



Figure 6: New About Us Page Design -- Cards

A card represents the container for the person's information. It contains an image of the person, their name, and their role as well as a brief description of their choosing.

Contained within each card is also a green button titled "More". Clicking this expands the user's card, providing a more detailed look into that person's biography seen in Figure 7.

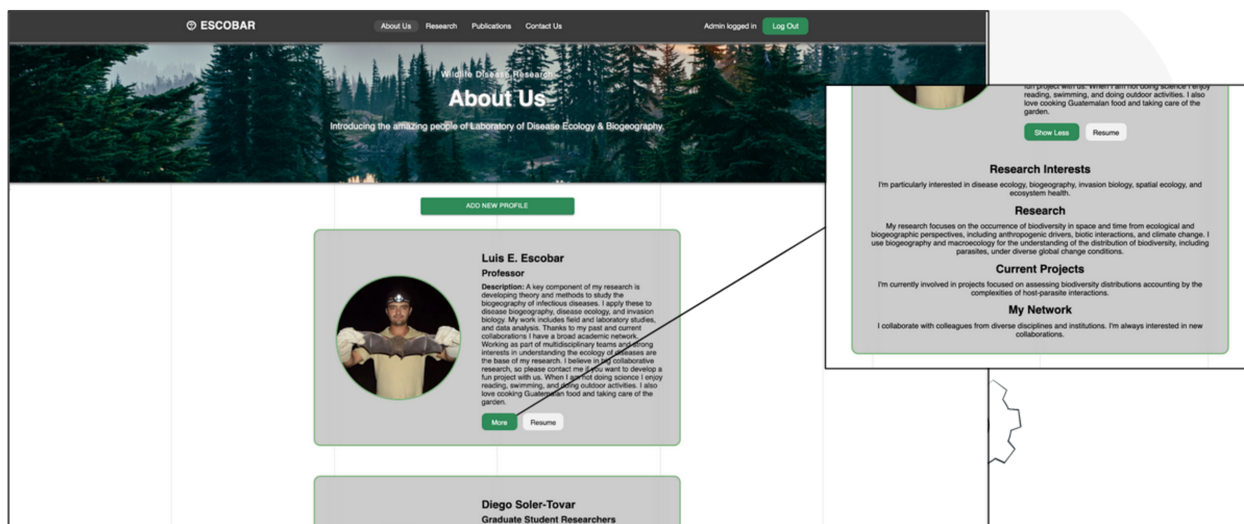



Figure 7: About Us Page – Expanded Card

The page is also scrollable, meaning the user can scroll through the list cards and select which one is of interest to them.

There is also a green "Add New Profile" button located above the cards. Should the user successfully login, then clicking the button will redirect to a page to create a new card shown in Figure 8.

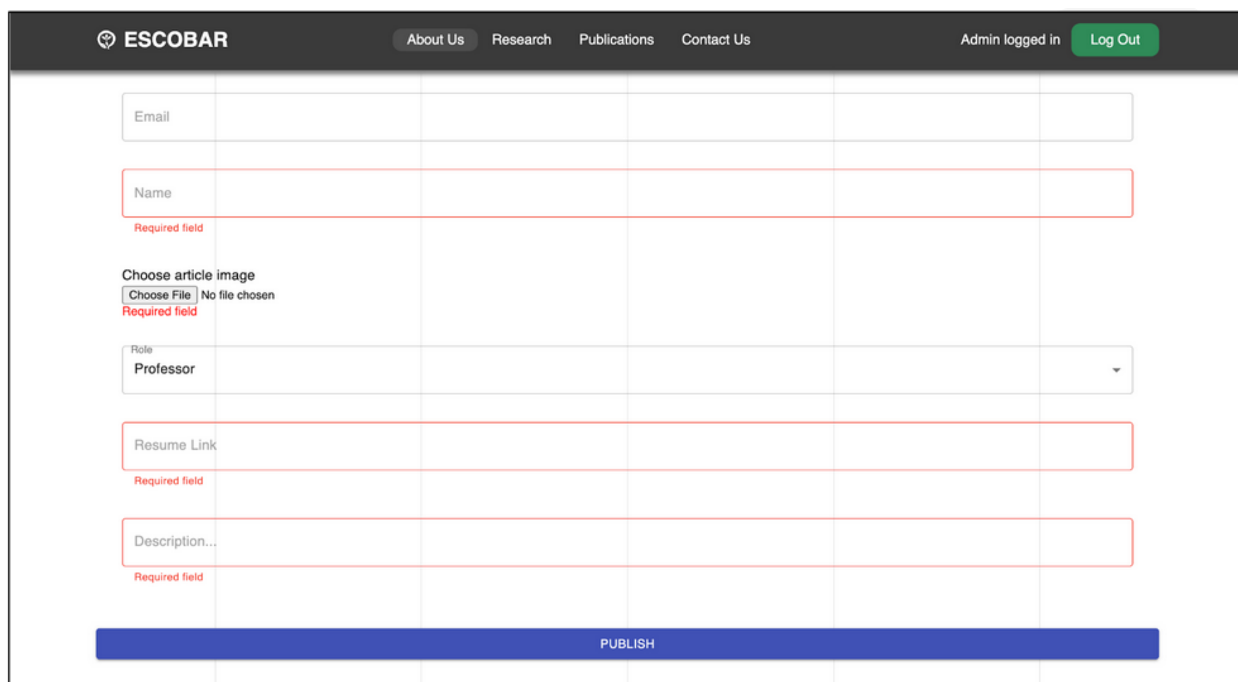



Figure 8: Add Card – About Us Page

The page contains a form with the following fields:

- Email
- Name
- Choose Article Image
- Role
- Resume Link
- Description

Filling out these fields and then clicking the blue "Publish" button will create a card that is displayed along with the other cards as seen in Figure 6.

## Publications Page

The Publications page serves as a hub for the laboratory's published works. They are organized in a card format containing the citation information for each work shown in Figure 9.

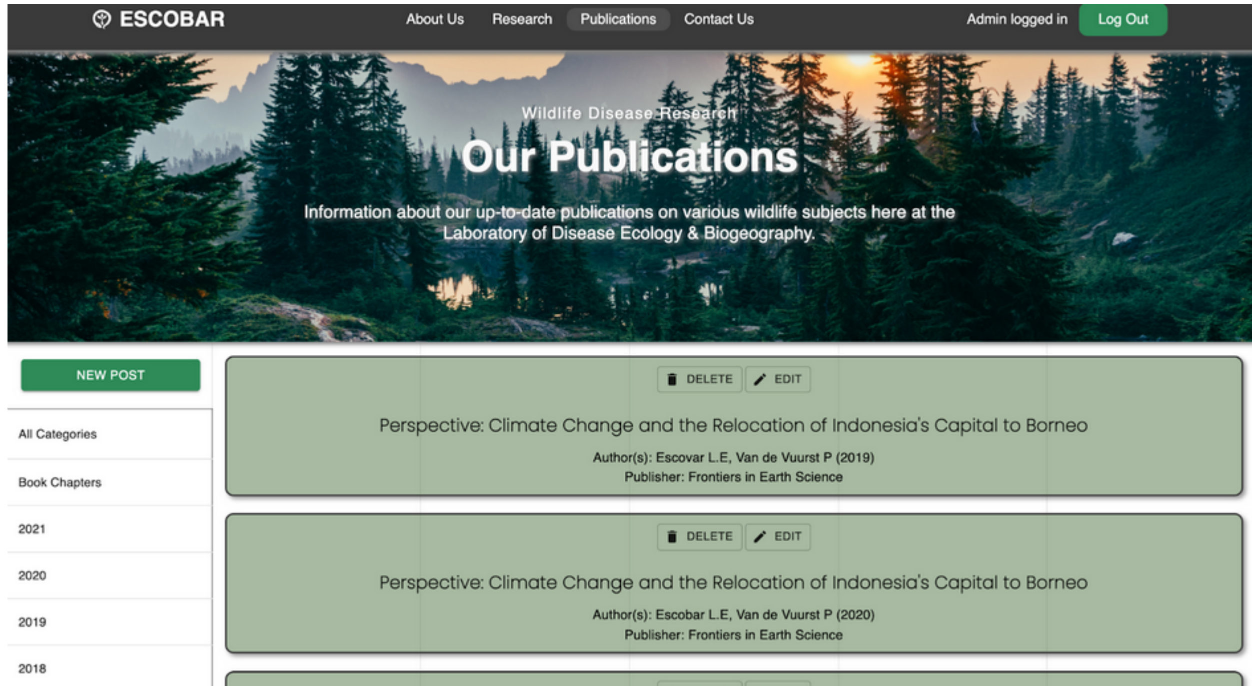



Figure 9: New Publications Page Design

The page is scrollable allowing for the user to scroll the list of publications and select the one of most interest. Another feature involves filtering categories to the left of the cards. They provide the user with the ability to select publications of a given year. That is, if the user clicks on "2021", then only publications from the year 2021 will be shown.

Also, the cards are clickable and, once clicked, the user is directed to the location on the web where that publication is contained.

Above the filtering category, there is a green "New Post" button. Provided a user is authenticated and logged in, clicking this button will allow the user to add a new publication to the list similar to that of Figure 9.

## Research Page

The Research page provides the user with a list of the research disciplines within the department. Each post is represented in a card format, with each card containing a description of the post's content as well as title, author, and content seen in Figure 10.

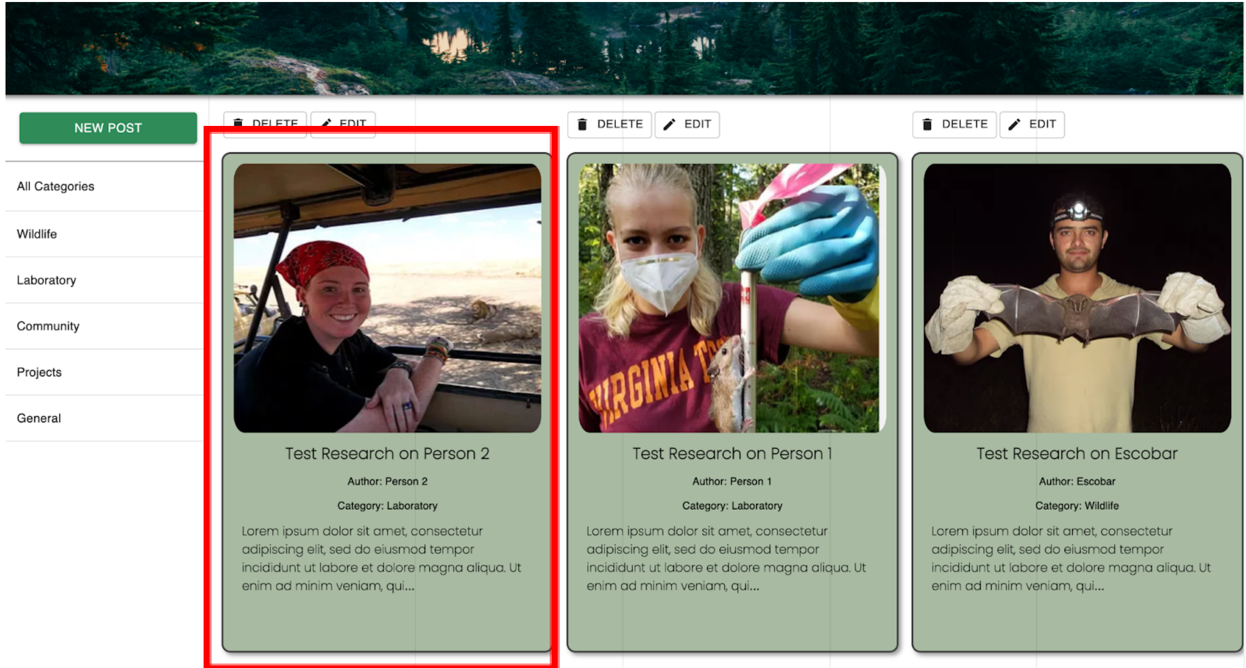



Figure 10: New Research Page Design

The cards are clickable, and upon clicking the card the user is redirected to a new page containing a more detailed view of the card shown in Figure 11.

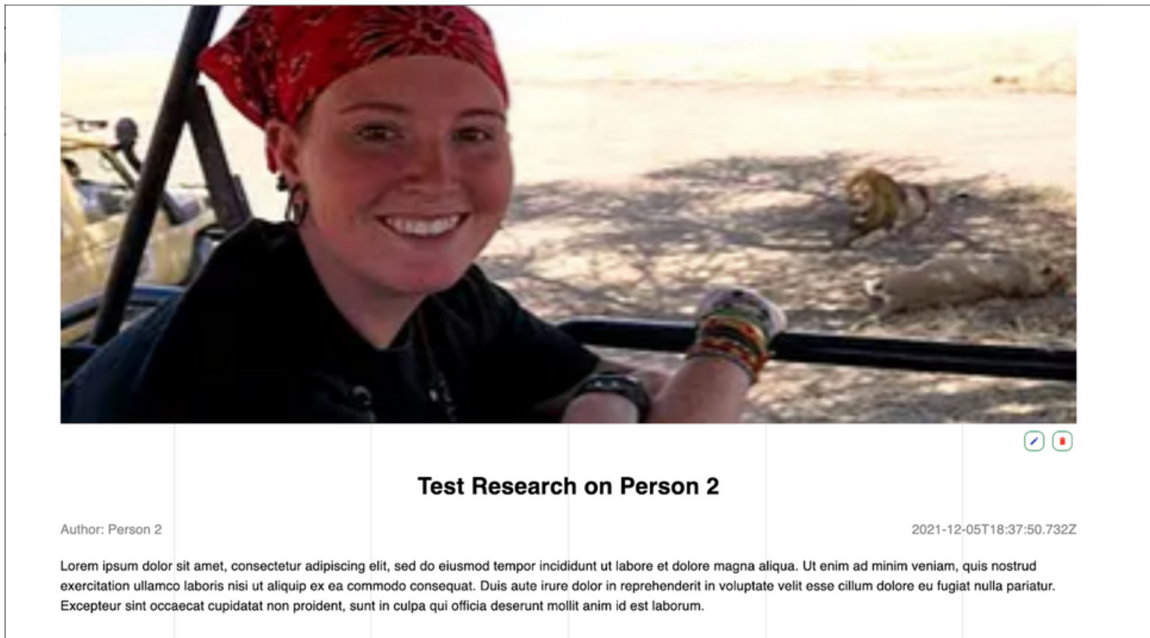



Figure 11: Research Page Card Expanded

Another feature of the research page is the ability to filter the posts based on category. To the left of the cards is a series of categories (see Figure 10); clicking any of the categories filters the cards on the right to match the category selected.

Provided a user is authenticated and logged in, clicking the "New Post" button above the filtering categories will allow the user to add a new entry to the list similar to that of Figure 10.

## Contact Page

The Contact Page provides a means for prospective students, researchers, professors, etc. to contact Professor Escobar, utilizing the free EmailJS service.

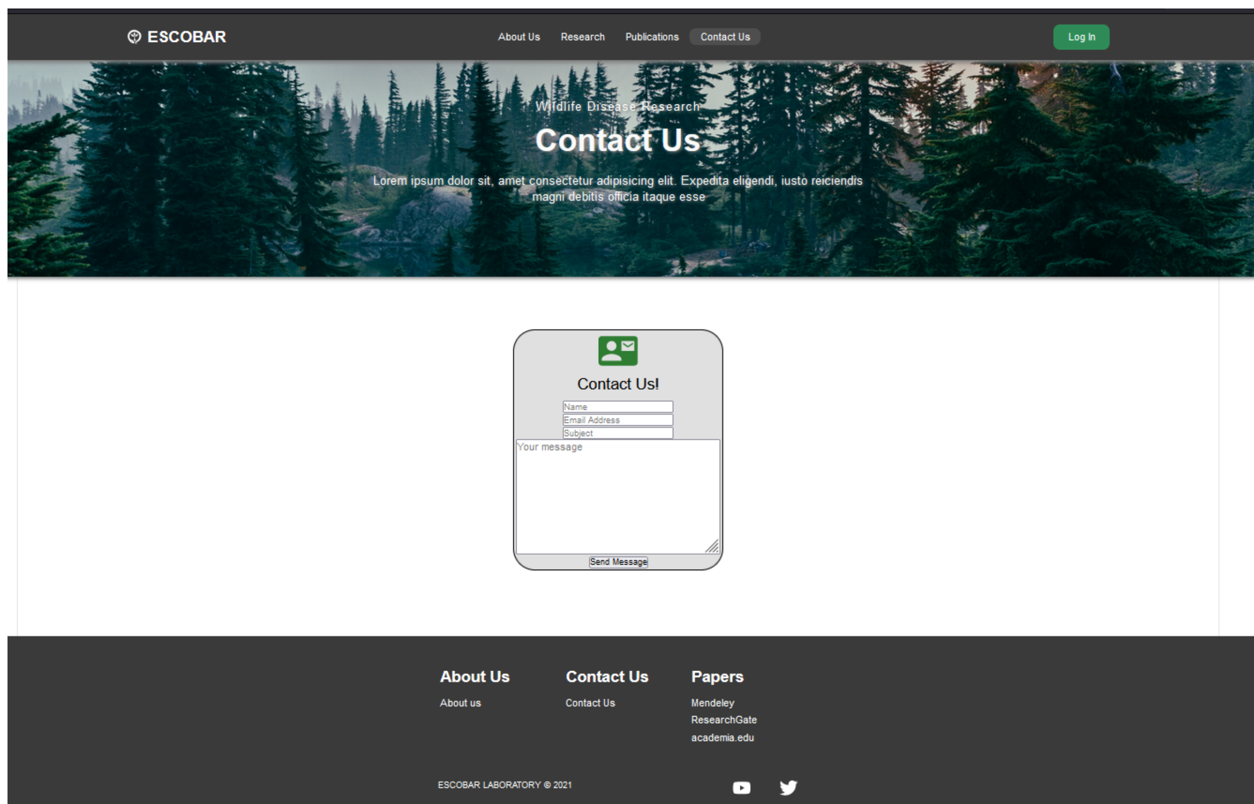



Figure 12: New Contact Us Page Design

The main component of the page is the form located in the center of the page shown in Figure 12 and 13.

Figure 13: Contact Us Page Form

There are four fields: name, email address, subject, and a message box. Once the user enters their name, a valid email address, a subject, and a message, they can click the “Send Message” button. Figure 14 illustrates a sample input form, while Figure 15 shows the corresponding email that is received:

Figure 14: Filled Out Contact Page Form

You got a new message from Kristopher Moratti:

Evening Professor Escobar, I was hoping you could let me know what the lecture on Friday was about. Thanks!

You can reach me at kris3@vt.edu.

Best wishes,

Kristopher Moratti

Email sent via EmailJS.com

Figure 15: Sample Email from Contact Form

## Administrator Login Page

The Administrator Login page is viewed after the user clicks the green "Log In" button in the header and can be seen in Figure 16.

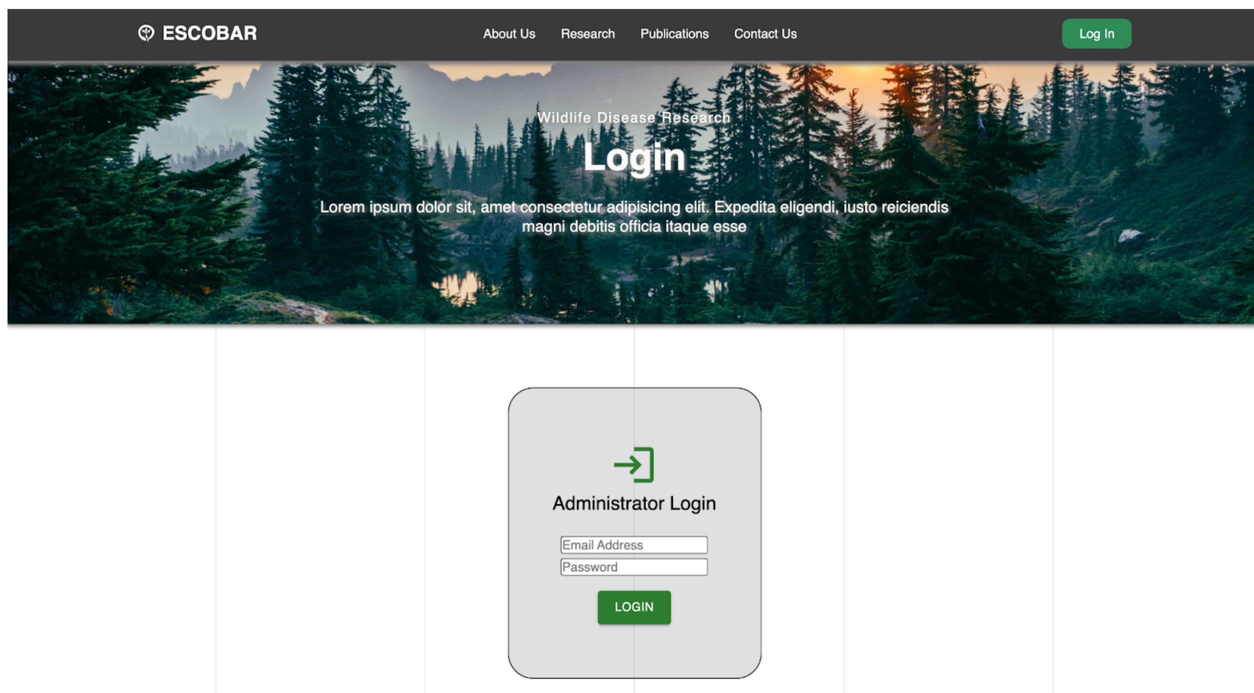



Figure 16: Administrator Login Page

The container located in the middle of the page contains two fields and a green "Login Button". If the proper login email and password are provided, then clicking the green "Login Button" will authenticate the user and provide access to the additonal buttons on the relevant pages.

# Developer's Manual

The Developer's Manual serves as an explanation of the code hierarchy while identifying the relevant files corresponding to front-end functionality and design aspects as well as back-end interaction.

The web application that we have created is constructed with a codebase architecture that is separated by the client and the server-side. The client part of the application consists of front-end React.js code that translates directly to the user interface and interaction components of the web application. Walking through the codebase on the client-side, we will explain top-level to bottom-level architecture.

## Application Layer

First, we have the App.js file located in the directory shown in Figure 17.

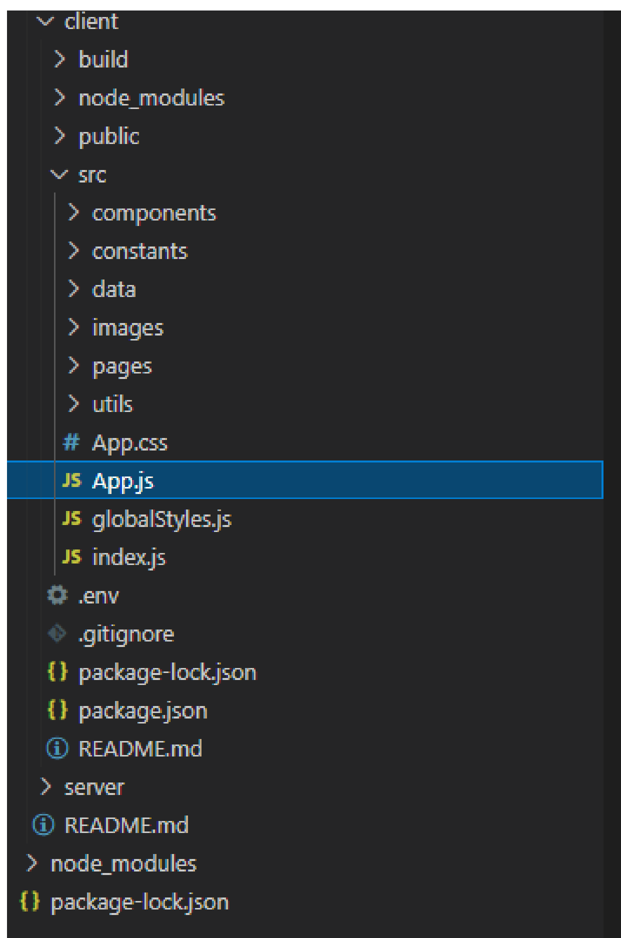



Figure 17: App.js File Location

This file's main purpose is to be used for routing that calls each page and necessary components which get rendered on the user's screen.

```
<Router>
  <GlobalStyle />
  <Sidebar isOpen={isOpen} toggle={toggle} />
  <Navbar toggle={toggle} />
  <Switch>
    <Route path="/" exact component={HomePage} />

    <Route path="/research" exact component={ResearchPage} />
    <Route path="/about" exact component={AboutPageDisplayer} />
    <Route path="/publications" exact component={PublicationsPage} />
    <Route path="/login" exact component={Login} />
    <Route path="/contact" exact component={ContactPage} />
    <Route path="/details/:id" exact component={DetailView} />
    <Route path="/research/create" exact component={ResearchCreateView} />
    <Route path="/research/edit/:id" exact component={ResearchEditView} />
    <Route path="/publications/create" exact component={PublicationCreateView} />
    <Route path="/publications/edit/:id" exact component={PublicationEditView} />
    <Route path="/about/create" exact component={AboutCreateView} />
    <Route path="/about/edit/:id" exact component={AboutEditView} />
    <Route path="/update" exact component={UpdateView} />
  </Switch>
</Router>
```

Figure 18: App.js Router Calls

As shown in Figure 18, routes are defined for each page/view, and have displayers so that each has its designated path in which the user can be directed based on the request from the user on the front-end. At a high-level glance, App.js can be considered as the most top layer of the entire front-end portion of the application. One important subject to mention is the Navbar and Sidebar components, which act as the controller on the user interface that allows the user to freely navigate throughout the website on their internet-compatible device. These components are contained within another directory.

## Pages Layer

Moving further down into the application layer, we have each page of the web application that the user can access shown in Figure 19.

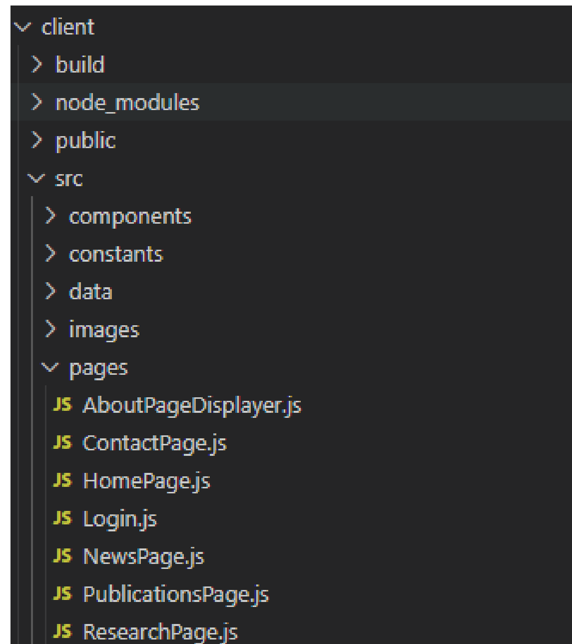



Figure 19: Pages Directory Location

```
function HomePage(prop) {
    return (
        <>
            <InfoSection {...homeObjOne} />
        </>
    );
}

export default HomePage;
```

Figure 20: Function for Displaying Home Page

This includes the Home page, Login page, About Us page, Research page, Publication page, and the Contact page. As mentioned in the Application Layer section, each page is associated with its own routing path and rendered accordingly by App.js. Based on the contents of each page, specific components and data are used to tailor the page to its intended functionalities as well as the information being displayed. An example function is shown in Figure 20 which renders the Home/Landing Page.

## Static Data Layer

Before moving further into lower-level layers, it is important to mention the static data that are frequently referenced on the front-end. The /data/data.js file contains data that is used simply for

displaying static information for each page description, which explains what the page provides for the user.

The /constants directory contains other static data that is used to enable categorized filtering of contents that are being displayed on the page. The /images directory contains .jpg files that may or may not be used to render images on the website. These directory locations are shown in Figure 21.

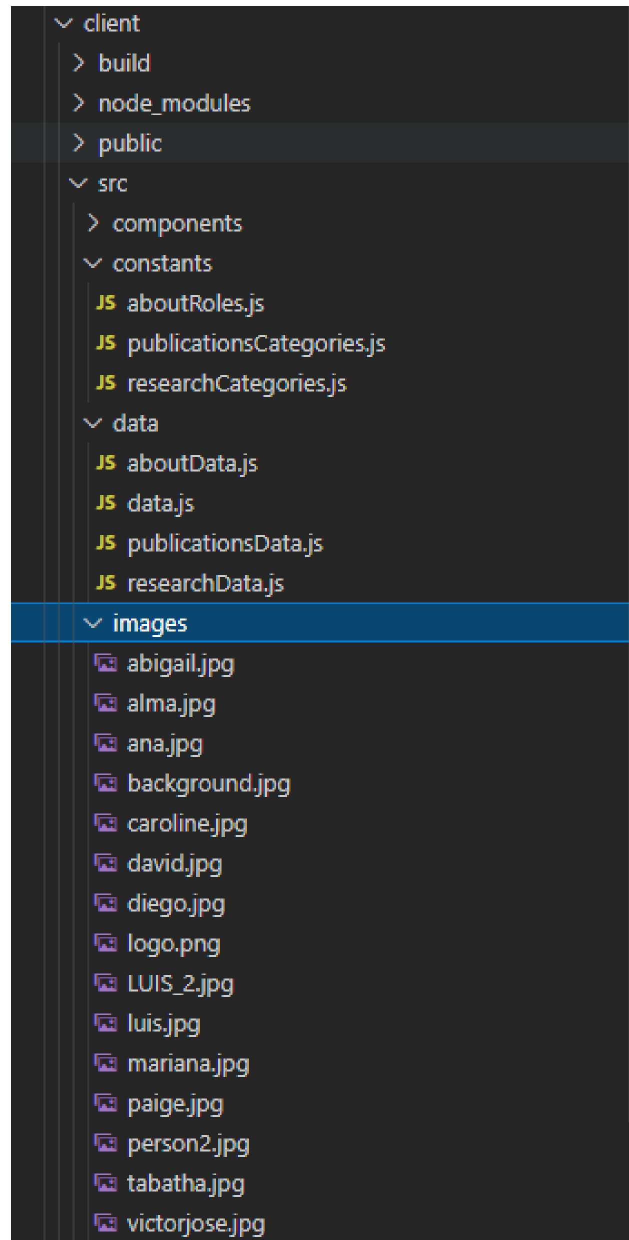



Figure 21: Static Data Directory Location

## Reusable Components Layer

The reusable components are the building blocks of our application. These components, seen in Figure 22,  are JavaScript classes or functions that are used to accept input data, have properties, and return UI elements that describe how the front-end should appear on the screen.

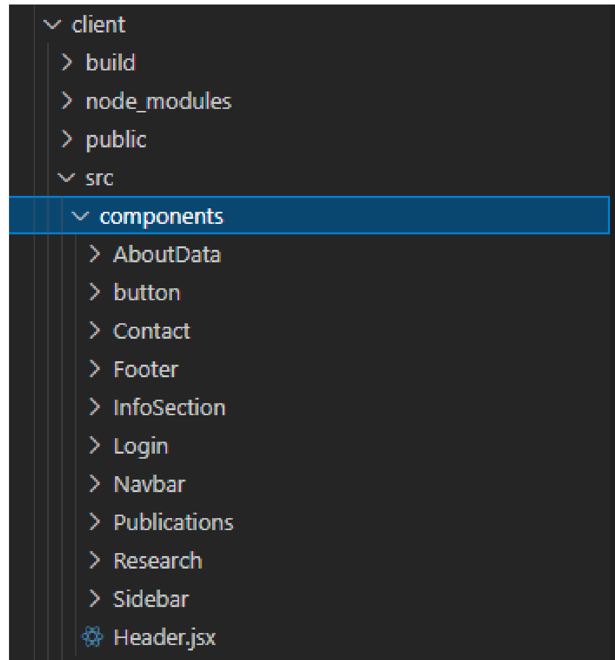



Figure 22: Component's Directory Location

Seen in Figure 23, the AboutData component further consists of AboutCreateView.js, AboutEditView.js, AboutLayout.js, AboutPagePosts.js, AboutPost.js, and AboutSinglePost.js.

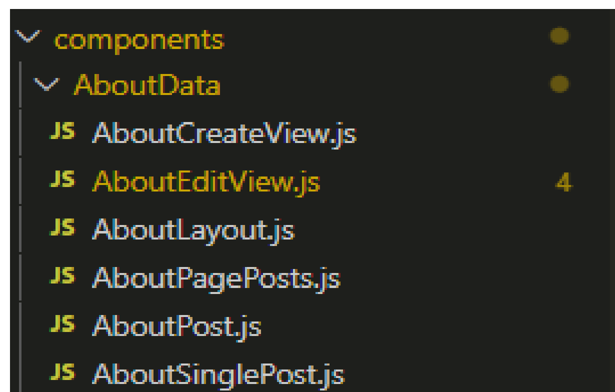



Figure 23: AboutData Directory Location

There are a total of ten reusable components:

- **AboutData**: Component consisting of the About Us page and input fields that can be edited or updated by the Admin user to add in new user information.
- **Contact**: Component consisting of the Contact Us page and input fields that can be used to send client's email and description to contact the Admin user.
- **Delete**: Component containing the delete icon and container with text to confirm the deletion.
- **Footer**: Component containing the Footer data, links to social media, and CSS styling.
- **InfoSection**: Component that takes in properties (data) from other components, and then arranges and displays them.
- **Login**: Component containing the Login Form data, LoginPage.js consisting of the input fields and LoginForm.js consisting of the call to the LoginForm.js page.
- **Navbar**: Component containing the Navigation bar details and its elements.
- **Publications**: Component containing the Publications page data. The Publications component further consists of:
    - post: consisting of the .jsx files PublicationCreateView and PublicationEditView to create and edit the Publication posts
- **Research**: Component containing the Research page data. The Research component further consists of:
    - post: consisting of the .jsx files ResearchCreateView and ResearchEditView to create and edit the Research posts
- **Sidebar**: Component containing the Sidebar data seen in the Research and Publications page.

## About Us Page Description

The content of the "About Us" page is generated by the AboutPageDisplayer.js file, shown in Figure 24, located in the pages directory as seen in Figure 19.

```
AboutPageDisplayer.js ×
client > src > pages > JS AboutPageDisplayer.js > [@] default
1  import React from "react";
2  import AboutLayout from "../components/AboutData/
   AboutLayout";
3  import InfoBanner from "../components/InfoSection/
   InfoBanner";
4  import { aboutObjOne } from "../data/data";
5
6  function AboutPageDisplayer(props) {
7    return (
8      <>
9        <InfoBanner {...aboutObjOne} />;
10       <AboutLayout />
11     </>
12   );
13 }
14
15 export default AboutPageDisplayer;
```

Figure 24: AboutPageDisplayer.js File

As shown in Figure 24, the file contains references to AboutLayout and InfoBanner components located on lines 9 and 10. These are imported from the components directory via lines 2 and 3. Line 4 imports aboutObjOne, Professor Escobar's about data-object imported from "../data/data" to be used in the InfoBanner component. Finally, the function AboutPageDisplayer() on line 6 returns the imported <InfoBanner> and <AboutLayout> components.

## Contact Us Page Description

The content of the "Contact Us" page is generated by the ContactPage.js file located in the pages directory as seen in Figure 19.

```
ContactPage.js
client > src > pages > ContactPage.js > ...
import React from 'react';
import InfoBanner from '../components/InfoSection/InfoBanner';
import { contactObjOne } from '../data/data';
import Contact from '../components/Contact/Contact';
function ContactPage(props) {
    return (
        <>
            <InfoBanner {...contactObjOne} />
            <Contact />
        </>
    );
}

export default ContactPage;
```

Figure 25: ContactPage.js File

The file shown in Figure 25 contains references to InfoBanner and Contact components located on lines 8 and 9. These are imported from the components directory via lines 2 and 4. On line 8, the InfoBanner utilizes a {...contactObjOne} data object as a property that is available via the import on line 3. Finally, the function ContactPage(props) defined in lines 5-11 returns <InfoBanner> and <Contact> components.

## Home Page Description

The Home/Landing Page display is controlled by the HomePage.js file, shown in Figure 26, located in the pages directory shown in Figure 19.

```
HomePage.js
client > src > pages > HomePage.js > ...
import React from 'react';
import InfoSection from '../components/InfoSection/InfoSection';
import { homeObjOne } from '../data/data';

function HomePage(prop) {
    return (
        <>
            <InfoSection {...homeObjOne} />
        </>
    );
}

export default HomePage;
```

Figure 26: HomePage.js File

The file shown in Figure 26 contains two important imports located on lines 2 and 3. These provide the file with access to the InfoSection component as well as the {homeObjOne} data object. The Function HomePage() returns the <InfoSection> component containing the entire layout of the home page. It takes {...homeObjOne} as a property.

## Login Page Description

The Login Page display is controlled by the Login.js file, shown in Figure 27, located in the pages directory.

```
JS Login.js  ×
client > src > pages > JS Login.js > [@] default
1   import React from 'react';
2   import LoginPage from '../components/Login/LoginPage';
3
4   import InfoBanner from '../components/InfoSection/
    InfoBanner';
5   import { loginObjOne } from '../data/data';
6
7
8   function Login() {
9       return (
10          <>
11              <InfoBanner {...loginObjOne} />
12              <LoginPage />
13          </>
14      );
15  }
16
17  export default Login;
```

Figure 27: Login.js File

This file contains references to InfoBanner and LoginPage components located on lines 11 and 12 which are made available via the imports on lines 2 and 4. There is also a reference to the {loginObjOne} data component in line 11 which is made available via the import on line 5. The Login() function defined in lines 8-15 returns the imported components.

## News Page Description

The layout of the News Page is controlled by the NewsPage.js file, shown in Figure 28, located in the pages directory.

```
NewsPage.js
client > src > pages > NewsPage.js > ...
1   import React from 'react';
2   import InfoBanner from '../components/InfoSection/
    InfoBanner';
3   import { newsObjOne } from '../data/data';
4
5   function NewsPage(props) {
6       return (
7           <>
8               <InfoBanner {...newsObjOne} />
9           </>
10      );
11  }
12
13  export default NewsPage;
```

Figure 28: NewsPage.js File

There are two important imports located at lines 2 and 3 that provide references to the InfoBanner component and the { newsObjOne } data component. The function defined on lines 5-11 returns the InfoBanner component and renders the data.

## Publications Page Description

The content of the Publications Page is displayed according to the PublicationsPage.js file shown in Figure 29.

```
JS PublicationsPage.js ×
client > src > pages > JS PublicationsPage.js > [@] default
1   import React from 'react';
2   import InfoBanner from '../components/InfoSection/
    InfoBanner';
3   import { publicationsObjOne } from '../data/data';
4   import Publications from '../components/Publications/
    Publications'
5
6   function PublicationsPage(props) {
7       return (
8           <>
9               <InfoBanner {...publicationsObjOne} />
10              <Publications />
11          </>
12      );
13  }
14
15  export default PublicationsPage;
```

Figure 29: PublicationsPage.js file

There are two components imported on lines 2 and 4: InfoBanner component and Publications component. Line 3 imports the {publicationsObjOne} data component which is used to define the InfoBanner component. The function PublicationsPage() returns the relevant components and renders the data accordingly.

## Research Page Description

The rendering of content for the Research page is controlled by the ResearchPage.js file shown in Figure 30.

```
JS ResearchPage.js ×
client > src > pages > JS ResearchPage.js > ...
1   import React from "react";
2   import Research from "../components/Research/Research";
3   import InfoBanner from '../components/InfoSection/
    InfoBanner';
4   import { researchObjOne } from '../data/data';
5
6   function ResearchPage(props) {
7     return (
8       <>
9         <InfoBanner {...researchObjOne} />
10        <Research />
11      </>
12    );
13  }
14
15  export default ResearchPage;
```

Figure 30: ResearchPage.js File

There are two components and one data component imported via lines 2, 3, and 4. These components are used in the ResearchPage function defined in lines 6-13. This function returns these components allowing the page to display the correct information.

## About Data Component

The following is a description of the relevant files, functions, constants, etc. contained within the About Data component directory.

- /components/AboutData:
    - AboutCreateView.js: (If the user is logged in as the Admin) Used to create a new About section of a user by the Admin. It imports Box, Typography, Button, Form, InputLabel, MenuItem, TextField from material-ui, a React component library.
        - const useStyle = makeStyles(theme) to create change or update the styles of the components seen in the page.
        - Function AboutCreateView() holds the form and required fields to create an About section. It further holds
            - isEmptyField() function which validates if a field is left empty.
            - Function onChangeFile() to set the filename to the name of the image in the article
            - Function validate() to validate the submission of the form
            - Function submit() to submit the form once all the details are filled in
                - The submit function calls axios.put(`/api/about/${postId}`, body) to put/save the edited details data into the MongoDB database once the submit button is clicked and the form is submitted.
                - The body is an object variable that includes all the edited details like email, name, role, resumeLink, and description of the user.
            - The formData object to send/post the request to the backend into "/api/about"

    - The return statement of the AboutCreateView() function containing the JSX code component that consists of the following:
        - <Box> with a class name of the container. It encloses the entire About Data of a single user as a box.
        - <form> the form in which the Admin User should provide details of any user being added.

        - The <form> tag further contains four <TextField> tags. Each <TextField> tag holds the fields related to the Name of the user being added, Email, Resume Link and Description
        - The <form> tag also contains <FormControl> to choose the role of the user being added.
        - The <form> tag also contains a <Button> component "Publish" which, when clicked, submits the form data to the database and displays the newly added User information on the webpage.

- AboutEditView.js: (If the user is logged in as the Admin) Used to Edit the created About Post section of a user. Imports roles from "../../constants/"

    o const useStyle = makeStyles(theme) to create change or update the styles of the components seen on the page.

    o Function const AboutEditView() holds the form and required fields to edit an About section of a user.

    - Variable const postId to get the location of the required About post to be edited.
    - useEffect(async() ) function
        - The useEffect(async()) further contains
            - getAboutPost(postId) to get the post that needs to be edited into a const res variable.
            - setAboutPost(res) to set the About Post with the newly edited data.
            - setEmail(res.email) to set the About Post with the newly edited email of the user.
            - setName(res.name) to set the About Post with the newly edited name of the user.
            - setRole(res.role) to set the About Post with the newly edited role of the user.
            - setResumeLink(res.resumeLink) to set the About Post with the newly edited Resume link of the user.
            - setDescription(res.description) to set the About Post with the newly edited description of the user.
    - Function isEmptyField() function which validates if a field is left empty.
    - Function validate() to validate the submission of the form
    - Function submit to submit the form once all the edited details are filled in:

    - The submit function calls axios.put(`/api/about/${postId}`, body) to put/save the edited details data into the database once the submit button is clicked and the form is submitted.
    - where "body" is an object variable that includes all the edited details like email, name, role, resumeLink, and description of the user.

- AboutLayout.js: Component dedicated to displaying the Layout of the About Us page.
    - const useStyle = makeStyles(theme) to create, change, or update the styles of the components seen on the page
    - Function AboutLayout() holds the code related to the Layout of the About Us page of the website.
        - AboutLayout() function sets aboutPosts and setAboutPosts using the useState hook to state the aboutPosts variable and the setAboutPosts variable.
        - AboutLayout() function makes use of useEffect() using async await to wait for the About Posts of each user from the MongoDB database into a variable 'res' called from the getAboutPostsFromDatabase() function.
        - The res variable is then passed to setAboutPosts() as a parameter to set the About Us posts.
        - The async function getAboutPostsFromDataBase() fetches the About posts of users from the database using the axios await and get method from the '/api/about' URL. It returns result.data.
        - The return statement of the About layout functional component contains the layout logic.
        - Initially, the token of authentication is checked and if the user is logged in as Admin, they can see an "Add New Profile" button that allows the Admin user to create a new About Profile of the user.

- AboutPagePosts.js: Component that sorts and displays all the About Us page posts based on roles: Professors, Graduate Students, Undergraduate Students, and Visiting Scholars.
    - AboutPagePosts imports AboutPost from './AboutPost' and user roles from "../../constants/aboutRoles"
    - The function AboutPagePosts sets the sortedAboutPosts variable to sortPostsByRole(props.aboutPosts).
    - The function sortPostsByRole() takes in aboutPosts as a parameter and returns the sorted About Posts by role.

        - The function sortPostsByRole() filters the posts into professors, gradStudents, undergradStudents and visitingScholars roles, sorts them and returns them in the sortedPostsByRole variable.
    - The function AboutPagePosts then maps over the sortedAboutPosts variable that was set and returns the About Posts UI containers <AboutPageContainer> with the related <AboutPost> component based on roles.

- AboutPost.js: The Component that sets the UI of a single About Post of the Admin. It sets the UI fields such as the Name of the Admin, Description, Research Interests, and Resume of the Admin. It displays the buttons "show more" and "edit."
    - AboutPost component imports Delete button from '../Delete/Delete'
        - The AboutPost() function takes in name, image, role, description, resumeLink, and ID properties. Sets the showMore and buttonLabel state hooks using useState().
        - The AboutPost() function returns the Card containing the UI layout with Admin details.
        - It also returns the Delete button, Link to Resume button, and Show More button, which when clicked, displays more data about the Admin.

- AboutSinglePost.js: The Component that manages the UI of a single About post.
    - Imports NavBtnForAboutPage button from "../Navbar/NavbarElements."
        - The function AboutSinglePost returns a Box containing the containers for name, role, department, college, description, More About Me, and My Resume buttons of a single user.

## Button Component

Figure 31 represents a detailed view of how the Button component is implemented.

```
import React from "react";
import styled from "styled-components";

const ButtonWrapper = styled.button`
  padding: ${({ small }) => (small ? "5px 9px" : "7px 15px")};
  border-radius: 5px;
  background-color: #4263f5;
  color: #fff;
  font-weight: bold;
  font-size: ${({ small }) => (small ? "12px" : "16px")};
  outline: none;
  border: 2px solid transparent;
  transition: all 220ms ease-in-out;
  cursor: pointer;

  &:hover {
    background-color: transparent;
    border: 2px solid #4263f5;
  }
`;

export function Button(props) {
  return <ButtonWrapper {...props}>{props.children}</ButtonWrapper>;
}

```

Figure 31: Button Component Implementation

The index.jsx file, shown in Figure 31, representing the Button component contains two important definitions which begin on lines 4 and 22, respectively. Line 4 defines the styling attributes of the button while the function Button() beginning on line 22 returns the button defined by the styling elements.

## Contact Page Component

The Contact Page functionality revolves around the EmailJS service. After logging in using the required credentials, EmailJS provides the ability to create email templates for different services as shown in Figure 32.

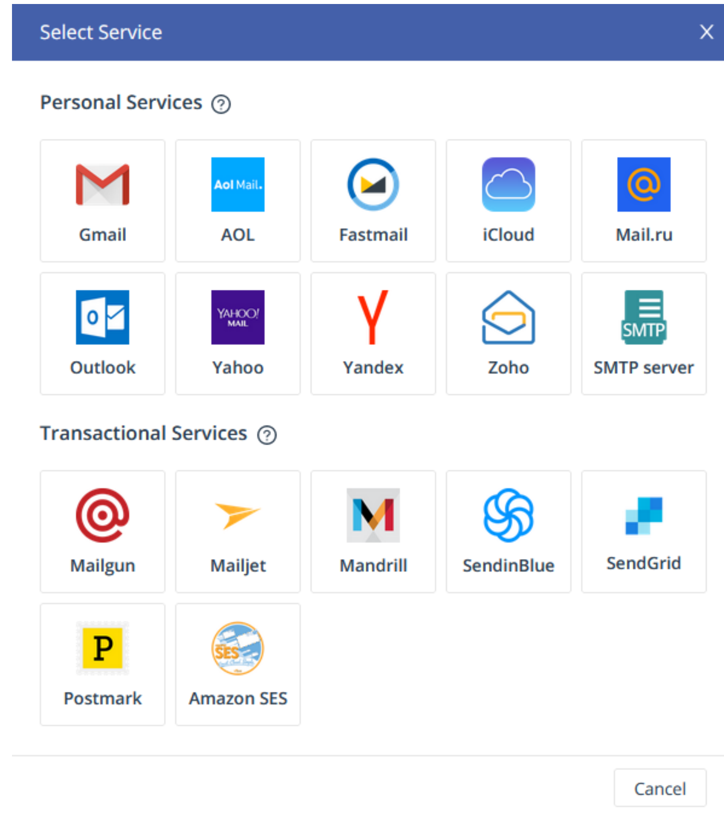



Figure 32: EmailJS Available Service

For example, selecting the Gmail service, which is the service used in the website, creates the entry shown in Figure 33 in the account under "Email Services".

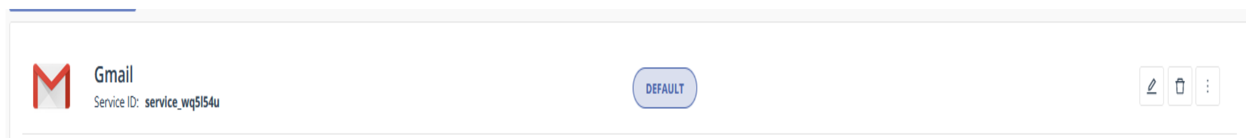



Figure 33: Example Service

The important thing to note is the Service ID which is used to send the form contents on the webpage to EmailJS. As for the implementation of this page, the front-end design and EmailJS functionality are contained in the Contact.js located in the Contact folder in the Components directory shown in Figure 34.

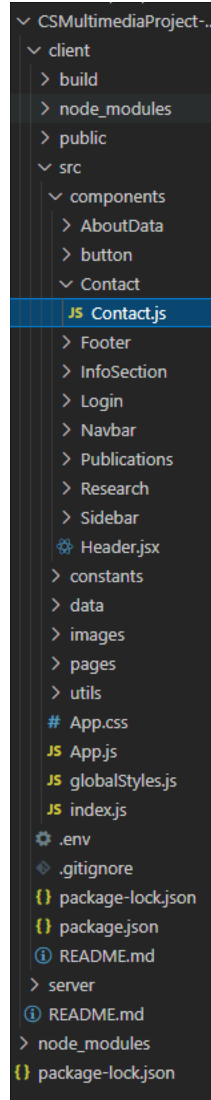



Figure 34: Location of Contact.js

The code located in the Contact.js file shown in Figure 35 represents the rendering of the form.

```
22      return (
23          <>
24              <ContactContainer>
25
26                  <form onSubmit={handleSubmit} style={containerStyle}>
27
28                      <div className="row pt-5 mx-auto">
29                          <ContactMailIcon color="success" sx={{ fontSize: 60 }} />
30                          <Typography variant="h5" gutterBottom component="div">
31                              Contact Us!
32                          </Typography>
33                          <div className="col-8 form-group mx-auto">
34                              <input type="text" className="form-control" placeholder="Name" name="name" />
35                          </div>
36                          <div className="col-8 form-group pt-2 mx-auto">
37                              <input type="email" className="form-control" placeholder="Email Address" name="email" />
38                          </div>
39                          <div className="col-8 form-group pt-2 mx-auto">
40                              <input type="text" className="form-control" placeholder="Subject" name="subject" />
41                          </div>
42                          <div className="col-8 form-group pt-2 mx-auto">
43                              <textarea className="form-control" id="" cols="30" rows="8" placeholder="Your message" name="message"></textarea>
44                          </div>
45                          <div className="col-8 pt-3 mx-auto">
46                              <input type="submit" className="btn btn-info" value="Send Message"></input>
47                          </div>
48                      </div>
49                  </form>
50              </ContactContainer>
51          </>
52      );
```

Figure 35: ContactPage.js – Form Definition

The form's structure is defined in lines 26 through 49. Within those lines, there are five elements of note: 3 input types, located at lines 34, 37, and 40; one text area; and one button, respectively, located at lines 34, 37, 40, 43, and 46. These elements represent the input fields as well as the button shown on the Contact Page. Should a user want to add or remove input fields, that should be done within these lines and in a format mirroring the defined structure. Furthermore, the styling of these form elements is done at the bottom of the Contact.js file contained within lines 55 and 76 as seen in Figure 36. These are merely styling elements controlling the aesthetics of the form and don't affect the functionality of each element.

```
55  const containerStyle = {
56      backgroundColor: 'rgba(100, 100, 100, 0.2)',
57      paddingLeft: 20,
58      paddingRight: 20,
59      borderStyle: 'solid',
60      borderWidth: '2px',
61      borderRadius: '2rem',
62      width: '20rem',
63      height: '23rem',
64      display: 'flex',
65      alignItems: 'center',
66      justifyContent: 'center'
67
68  }
69
70  const ContactContainer = styled.div`
71      display: flex;
72      align-items: center;
73      justify-content: center;
74      text-align: center;
75      margin-top: 5rem;
76  `
```

Figure 36: ContactPage.js – Styling Elements

To finish, EmailJS needs a template to properly interpret the fields defined in the form. With EmailJS, the logged-in user can create a new, editable template. Once created, the template is saved to the user's account and has an ID created as seen in Figure 37.

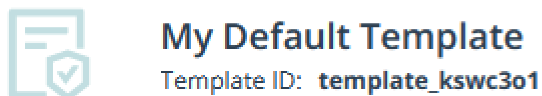



Figure 37: Sample Template

Once a template is created, the logged in user can edit and create a template of their desire. The template must be able to reference the fields contained in the form. For each input field, such as the one shown in Figure 38, there is a name attribute.

```
34                      <input type="text" className="form-control" placeholder="Name" name="name" />
```

Figure 38: Form Input Field Example

In this case, the name attribute equals “name”. The template then references this value by enclosing the value in triple brackets, such as {{{name}}} shown in Figure 39. It is important that what is enclosed in the triple brackets is equivalent to whatever the name attribute equals.

Figure 39: Email Template

The template references 4 separate variables that correspond to the input types defined in the form container seen in Figure 35. For an example of input and output, reference Figures 14 and 15.

There is also a formHandleSubmit() function defined shown in Figure 40 that is called when the “Submit Message” button is clicked.

```
10    function handleSubmit(e) {
11
12        e.preventDefault();
13        emailjs.sendForm("service_id", 'template_id', e.target, 'unique_user_id')
14            .then((result) => {
15                console.log(result.text);
16            }, (error) => {
17                console.log(error.text);
18            });
19        e.target.reset();
20
21    };
```

Figure 40: Form Submit Function -- Contact.js

The most important line to note is line 13 in Figure 40 which sends the form values to the EmailJS service and creates the email. The parameters are the service id, template-id, e.target, and the User id all of which are discerned through the EmailJS website UI.

## Footer Component

The Footer Component is fully implemented using the following files:

- Footer.css
- Footer.js
- FooterElement.js
- FooterFunction.js

Footer.css contains the styling attributes for each of the elements involved in creating the Footer. Any aesthetic changes to the Footer would be done by editing the styling fields in this file.

Footer.js represents the final rendering of the Footer component and is shown in Figure 41.

```
import React from 'react';
import Footer from './FooterFunctions'
import TwitterIcon from '@mui/icons-material/Twitter';
import YouTubeIcon from '@mui/icons-material/YouTube';

export function FooterContainer() {
    return (
        <Footer>
            <Footer.Wrapper>
                <Footer.Row>
                    <Footer.Column>
                        <Footer.Title>About Us</Footer.Title>
                        <Footer.Link href="#">Escobar</Footer.Link>
                        <Footer.Link href="#">Lab Alumni</Footer.Link>
                        <Footer.Link href="#">Visiting Scholars</Footer.Link>

                    </Footer.Column>
                    <Footer.Column>
                        <Footer.Title>Papers</Footer.Title>
                        <Footer.Link href="#">Medeley</Footer.Link>
                        <Footer.Link href="#">ResearchGate</Footer.Link>
                        <Footer.Link href="#">academia.edu</Footer.Link>

                    </Footer.Column>
                    <Footer.Column>
                        <Footer.Title>Contact</Footer.Title>
                        <Footer.Link href="#">Message Us</Footer.Link>
                        <Footer.Link href="#">Sign Up</Footer.Link>

                    </Footer.Column>
                    <Footer.Column>
                        <Footer.Title>Social</Footer.Title>
                        <Footer.Link href="#"><TwitterIcon fontSize="small" /></Footer.Link>
                        <Footer.Link href="#"><YouTubeIcon fontSize="small" /></Footer.Link>

                    </Footer.Column>
                </Footer.Row>
            </Footer.Wrapper>
        </Footer>

    )
}
```

Figure 41: Footer.js File

The function FooterContainer() shown in Figure 41 generates and returns a Footer element. Each <Footer.Column> tag corresponds to a different section in the Footer. The <Footer.Title> tags represent the section headers and the <Footer.link> tags represent the smaller text underneath the section headers.

FooterElements.js provides styling definitions by defining a series of styled constants seen in Figure 42.

```
3  export const Container = styled.div`
4      padding: 20px 60px;
5      margin-top: 100px;
6      background: radial-gradient(circle, rgba(58, 58, 58, 1) 0%, rgba(58, 58, 58, 1) 100%);
7      bottom: 0;
8  `
```

Figure 42: FooterElements.js Example Styled Constant

These control the aesthetics as well as the arrangement of the footer contents on each page.

FooterFunction.js provides the definitions for the different tags used in creating the Footer element in Footer.js.

```
1  import React from 'react';
2  import { Container, Wrapper, Row, Column, Link, Title } from './FooterElements'
3
4  export default function Footer({ children, ...restProps }) {
5      return <Container {...restProps}>{children}</Container>
6
7  }
8
9  Footer.Wrapper = function FooterWrapper({ children, restProps }) {
10     return <Wrapper {...restProps}>{children}</Wrapper>
11 }
12
13 Footer.Row = function FooterRow({ children, restProps }) {
14     return <Row {...restProps}>{children}</Row>
15 }
16
17 Footer.Column = function FooterColumn({ children, restProps }) {
18     return <Column {...restProps}>{children}</Column>
19 }
20
21 Footer.Link = function FooterLink({ children, restProps }) {
22     return <Link {...restProps}>{children}</Link>
23 }
24
25 Footer.Title = function FooterTitle({ children, restProps }) {
26     return <Title {...restProps}>{children}</Title>
27 }
```

Figure 43: FooterFunction.js

Each definition shown in Figure 43 provides a rendering of a different aspect of the footer. For example, Footer.row defines how the text underneath each section header is arranged. InfoBanner Component

## InfoBanner Component

The InfoBanner component is defined by the following files:

- InfoBanner.js
- InfoBannerElement.js
- InfoSection.js

- InfoSectionElement.js

InfoBanner.js is the class used to produce the final rendering of the component. It contains a function InfoBanner() that returns this rendering using the constants defined in InfoBannerElement.js.

InfoBannerElement.js and InfoSectionElement.js represent the styling of each element used in this component. They describe the aesthetics as well as the physical arrangement of each element. These constants are defined similarly to those shown in Figure 42.

InfoSection.js contains a single function InfoSection() that produces the final rendering of the component using the constants defined in InfoSectionElement.js.

## Login Component

The Login component contains 3 files:

- LoginForm.js
- LoginPage.css
- LoginPage.js

LoginPage.css is a styling sheet used to define the aesthetics and physical arrangements of the relevant elements used to create the Login page.

LoginPage.js has a single LoginPage() that produces the final rendering of the component defined and styled using LoginForm.js and LoginPage.css.

LoginForm.js contains an important handleFormSubmit() function shown in Figure 44.

```
43    const handleFormSubmit = async (e) => {
44        e.preventDefault();
45        const credentials = { email, password };
46        await login(credentials);
47    }
48
```

Figure 44: handleFormSubmit() function

The handleFormSubmit() function is what is called when a user attempts to login. It reads the values entered into the email and password fields, and then calls the login() function shown in Figure 45.

```
async function login(credentials) {
    try {
        const res = await axios.post("/api/auth", credentials);
        const jwt = res.data.token
        localStorage.setItem("token", jwt);
        window.location.href = "/";
    }
    catch (e) {
        console.error("ERROR", e);
        localStorage.removeItem("token");
        setValid(false);
    }
}
```

Figure 45: login() Function

The login function reads the credentials provided by the form and validates them. This provides user authentication for the website.

These functions are defined within the LoginForm() function which also is responsible for rendering the content displayed on the Login page.

### Navbar Component

The Navbar component is created using two files:

- Navbar.js
- NavbarElement.js

NavbarElement.js defines a series of constants similarly to those in Figure 38. These constants represent the elements present in the Navbar component as well as their design aspects.

Navbar.js has a single function shown in Figure 46 that returns the rendering of these components.

```
5  function Navbar({ toggle }) {
6      return (
7          <>
8              <Nav>
9                  <NavLogo to="/">
10                     <NavIcon />
11                     ESCOBAR
12                 </NavLogo>
13                 <MobileIcon onClick={toggle}>
14                     <FaBars />
15                 </MobileIcon>
16                 <NavMenu>
17                     <NavLink to="/about" activeStlye>
18                         About Us
19                     </NavLink>
20                     <NavLink to="/research" activeStlye>
21                         Research
22                     </NavLink>
23                     <NavLink to="/publications" activeStlye>
24                         Publications
25                     </NavLink>
26                     <NavLink to="/contact" activeStlye>
27                         Contact Us
28                     </NavLink>
29                 </NavMenu>
30                 {!localStorage.token && (
31                     <NavBtn>
32                         <NavBtnLink to="/login">Log In</NavBtnLink>
33                     </NavBtn>
34                 )}
35                 {localStorage.token && (
36                     <NavBtn>
37                         <div style={{ color: "white" }}>Admin logged in</div>
38                         <NavBtnLink to="/" onClick={() => logout()}>
39                             Log Out
40                         </NavBtnLink>
41                     </NavBtn>
42                 )}
43             </Nav>
44         </>
45     );
46 };
```

Figure 46: Navbar.js Function

Each tag, such as <NavLogo> or <NavLink>, is defined in the NavBarElement.js.

## Sidebar Component

The Sidebar component is created using two files:

- Sidebar.js
- SidebarElement.js

Sidebar.js contains a single function that returns the rendering of the elements defined as constants in SidebarElement.js. The structure of these files is similar to that of the files present in the Navbar component.

## Publications Component

The Publications Page utilizes the Publications component which can be described as follows:

- Publications Component consists of a “post” component used to create and edit publication posts.
- PubicationPost.js component defines the UI layout of a single Publication Post.
- Publications.js component arranges all the Publications by fetching them from the database and filtering them by year.
- PublicationsPosts.js component assigns values and arranges the Publications’ cards into a grid.

The file located at Publications->post->PublicationCreateView.jsx

- PublicationCreateView.jsx: function PublicationCreateView() is used to create, validate, and submit a single publication post to the MongoDB database
- Function createRange(start, end) creates a Range of publication post results dating from 2011 till 2025, mapping over each year
- Function isEmptyField() returns if the field is left empty
- Function validate() validates if all of the fields are filled with data
- Function submit() submits the post to the MongoDB database using axios post if the post is validated

The file located at Publications->post->PublicationEditView.jsx

- PublicationEditView.jsx: function PublicationEditView() is used to edit a given publication post, from the postId, when the user is logged in as an admin
- Function createRange(start, end) creates a Range of publication post results dating from 2011 till 2025, mapping over each year
- Function isEmptyField() returns if the field is left empty
- Function validate() validates if all of the edited fields are filled with data
- Function submit() submits the edited post to the MongoDB database using axios post if the edited post is validated - to /api/publications.

The file PublicationPost.js shown in Figure 47 represents the part of the Publications component that contains individual Publication post components. It defines the UI layout of a single Publication Post.

- The PublicationPost() function returns a <BoxContainer> box that does the following:
    - Deletes the publication post using the Delete button, only if the user is logged in as the Admin
    - Edits the publication post using the Edit button, only if the user is logged in as the Admin.
    - Displays Publication post data like author and publishing year.

```
PublicationPost.js
client > src > components > Publications > PublicationPost.js > PublicationsPost
50  });
51
52  //Individual Post component
53  const PublicationsPost = ({ title, author, publisher, year, link, id }) => {
54      const classes = useStyles();
55
56      return (
57        <BoxContainer>
58          <Box className={classes.container}>
59            {localStorage.token && (
60              <div className={classes.buttonContainer}>
61                <Delete deleteEndpoint={`/api/publication/${id}`} />
62                <Link
63                  to={`/publications/edit/${id}`}
64                  style={{ textDecoration: "none", color: "inherit" }}
65                >
66                  <Button
67                    variant="outlined"
68                    startIcon={<EditIcon />}
69                    size="small"
70                    onClick={() => {
71                      console.log("edit");
72                    }}
73                  >
74                    Edit
75                  </Button>
76                </Link>
77              </div>
78            )}
79            <a
80              href={link}
81              style={{
82                textDecoration: "none",
83                color: "inherit",
84                textAlign: "center",
85              }}
86            >
```

Figure 47: PublicationPost.js Function

The file Publications.js represents the part of the component that arranges all of the Publications by fetching them from the database and filtering them by year.

- The function Publications() fetches publication posts from the database using getPublicationPostsFromDatabase() in useEffect shown in Figure 48. It sets the publication posts to a state hook using setPublicationPosts().
    - function getPublicationPostsFromDatabase() uses the axios ‘get’ method to get posts from ‘/api/publication.’ URL.
    - function filterPublicationPostsFromDatabase(year) filters and posts publication posts to the database based on year using the axios ‘post’ request.
- The function then returns a Grid containing the data related to all the publication posts in a UI layout.

```
31  async function getPublicationPostsFromDatabase() {
32    const res = await axios({
33      method: 'get',
34      url: "/api/publication",
35      headers: { "Content-Type": "application/json" },
36    });
37    return res.data;
38  }
39
40  async function filterPublicationPostsFromDatabase(year) {
41    let body = { year: "" };
42    if (year) {
43      body.year = year;
44    }
45    const res = await axios({
46      method: "post",
47      url: "/api/publication/year",
48      headers: { "Content-Type": "application/json" },
49      data: body,
50    });
51    return res.data;
52  }
53
54  export default Publications;
```

Figure 48: get and filter publications functions

The file PublicationsPosts.js represents the component that assigns values and arranges PublicationsPost component (layout) into a grid.

- The function PublicationsPosts() shown in Figure 49 uses the fetched “post” property to map over the publicationPosts and arrange them into a Grid layout, calling the <PublicationsPost> component.

PublicationsPosts.js

client > src > components > Publications > PublicationsPosts.js > ...

```
import { Grid } from "@material-ui/core";
import PublicationsPost from "./PublicationPost";

//Posts component
const PublicationsPosts = (props) => {
    return props.publicationPosts.map((post) => (
        <Grid item md={12} sm={12} xs={12}>

            <PublicationsPost
                title={post.title}
                author={post.author}
                publisher={post.publisher}
                year={post.year}
                link={post.link}
                id={post._id}
            />

        </Grid>
    ));
};

export default PublicationsPosts;
```

Figure 49: PublicationsPosts Component

## Research Component

The Research Component represents the UI interface rendered on the Research page. It is defined by the following files:

- Research Component consists of a “post” component to create and edit Research posts
- ResearchPost.js component defines the UI layout of a single Research Post.
- ResearchPosts.js component arranges all the Research posts by fetching them from the database and filtering them based on categories.
- Categories.js component arranges, maps, and enables filter functionality of all the categories of Research posts.
- Research.js component assigns values and arranges the ResearchPosts component and Categories component (layout) into a grid.

The file located at Research->post->ResearchCreateView.jsx contains the following relevant functions and values:

- ResearchCreateView.jsx: function ResearchCreateView() is used to create, validate, and submit a single research post to the MongoDB database
- Creates formData data object to append to the database, and posts the formData to the database using axios post to '/api/research'
- Function onChangeFile() sets the file name.
- Function isEmptyField() returns if the field is left empty.
- Function validate() validates if all of the fields are filled with data.
- Function submit() submits the post to the MongoDB database using an axios post if the post is validated
- Returns a Box containing the required text fields and form controls of various fields in the Research Page.

The file located at Research->post->ResearchEditView.jsx contains the following relevant function, fields, and definitions.

- ResearchEditView.jsx: function ResearchEditView() is used to edit a given research post, from the postId, when the user is logged in as an admin
- Function getResearchPost(id) gets the required research post from the post ID using the axios get method from '/api/research/posts/${id}' URL.
- Function isEmptyField() returns if the field is left empty.
- Function validate() validates if all of the edited fields are filled with data.
- Function submit() submits the edited post to the MongoDB database using the axios post method, if the edited post is validated - to /api/research/${postId}'.
- The return statement returns a Box containing all of the UI components like TextFields, FormControls and Buttons -- of the Research posts that are to be edited if the user is logged in as the Admin or has the permissions to edit.

The file located at Research->post->DetailView.jsx provides the following functionality:

- DetailView() function is used to fetch the details of research posts to pull up a detailed view of the selected post.
- Uses state hook setResearchPost into researchPost.
- Uses useEffect to asynchronously fetch the research post data using getResearchPost(postId) function and sets the result to setResearchPost(result).
    - getResearchPost(postId) uses the axios get method to get the selected post ID data from the '/api/research/post/${id}' URL.
- The return statement returns a Box layout of the entire fetched and detailed view of the selected post.

The Categories.js file shown in Figure 50 represents the component that defines the categories of the Research Posts and their layout in the UI of the webpage.

- Categories() function uses categories, setPosts, filterPostsFromDatabase, getPostsFromDatabase, and createLink props to categorize the research posts, filter and fetch them accordingly.
- Function update(category) updates the research page with only the research posts from the selected category.
- Function allCategories() updates the research page with all of the categories.
- The return statement returns a table consisting of all the categories of the research posts and a link to New Post to enable the creation of a new research post.

```
lient > src > components > Research > JS Categories.js > useStyles
42  const Categories = ({ categories, setPosts, filterPostsFromDatabase,
    getPostsFromDatabase, createLink }) => {
43    const classes = useStyles();
44    return (
45      <>
46        {localStorage.token && (
47          <Link to={createLink} className={classes.link}>
48            <Button variant="contained" className={classes.create}>
49              New Post
50            </Button>
51          </Link>
52        )}
53
54        <Table className={classes.table}>
55          <TableHead>
56            <TableRow className={classes.row}>
57              <TableCell onClick={() => allCategories()}>
58                All Categories
59              </TableCell>
60            </TableRow>
61          </TableHead>
62          <TableBody>
63            {categories.map((c) => (
64              <TableRow className={classes.row}>
65                <TableCell onClick={() => update(c)}>{c}</TableCell>
66              </TableRow>
67            ))}
68          </TableBody>
69        </Table>
70      </>
71    );
72
73    async function update(category) {
74      let filteredPosts = await filterPostsFromDatabase(category);
75      setPosts(filteredPosts);
76    }
77
78    async function allCategories() {
79      const allPosts = await getPostsFromDatabase();
80      setPosts(allPosts);
81    }
82  };
```

Figure 50: Categories Component of Research

The ResearchPosts.js file shown in Figure 51 represents the component that arranges all of the Research posts by fetching them from the database and filtering them based on categories.

- Imports ResearchPost Component from './ResearchPost.'
- Function ResearchPosts() uses props to map over the researchPosts and arranges them into a Grid layout using the ResearchPost component.

```
client > src > components > Research > ResearchPosts.js > ...
1  import React from "react";
2  import { Grid } from "@material-ui/core";
3  import ResearchPost from "./ResearchPost";
4
5  //Posts component
6  const ResearchPosts = (props) => {
7    console.log(props.researchPosts)
8    return props.researchPosts.map((post) => (
9      <Grid item md={4} sm={6} xs={12}>
10
11       <ResearchPost
12         title={post.title}
13         author={post.author}
14         text={post.text}
15         img={post.img}
16         category={post.category}
17         id={post._id}
18       />
19     </Grid>
20   ));
21 };
22
23 export default ResearchPosts;
```

Figure 51: ResearchPosts Component

The file Research.js shown in Figure 52 represents the component that assigns values and arranges the ResearchPosts component and Categories component (layout) into a grid.

- The function Research() fetches research posts from the database using getResearchPostsFromDatabase() in useEffect. It sets the research posts to a state hook using setResearchPosts().
  - function getResearchPostsFromDatabase() uses the axios 'get' method to get posts from the '/api/research.' URL.
  - function filterResearchPostsFromDatabase(category) filters and posts research posts to the database based on year using the axios 'post' request to '/api/research/category' URL.

- The function then returns a Grid containing the data related to all of the research posts in a UI layout.

```
JS Research.js 1 ×
client > src > components > Research > JS Research.js > ...
 8
 9  const Research = () => {
10    const [researchPosts, setResearchPosts] = useState([]);
11    useEffect(async () => {
12      const res = await getResearchPostsFromDatabase();
13      setResearchPosts(res);
14    }, []);
15
16    return (
17      <>
18        <Grid container>
19          <Grid item md={2} xs={12} sm={2}>
20            <Categories categories={categories} setPosts={setResearchPosts}
            getPostsFromDatabase={getResearchPostsFromDatabase}
21              filterPostsFromDatabase={filterResearchPostsFromDatabase} createLink={"/
              research/create"}/>
22          </Grid>
23          <Grid container item md={10} xs={12} sm={10}>
24            <ResearchPosts researchPosts={researchPosts}/>
25          </Grid>
26        </Grid>
27      </>
28    );
29  };
30
31  async function getResearchPostsFromDatabase() {
32    const res = await axios({
33      method: 'get',
34      url: "/api/research",
35      headers: {"Content-Type": "application/json"},
36    });
37    return res.data;
38  }
```

Figure 52: Research Component

The file ResearchPost.js shown in Figure 53 represents the component that contains individual Research post components. It defines the UI layout of a single Research Post.

- The ResearchPost() function returns a <BoxContainer> box that does the following:
    - Deletes the research post using the Delete button, only if the user is logged in as the Admin
    - Edits the research post using the Edit button, only if the user is logged in as the Admin.
    - Displays Research post data like title, author, image, category, and a brief description of the post.

```
ResearchPost.js ×
client > src > components > Research > ResearchPost.js > ResearchPost
63  //Individual Post component
64  const ResearchPost = ({ title, author, category, text, img, id }) => {
65    const classes = useStyles();
66
67    return (
68      <BoxContainer>
69        {localStorage.token && (
70          <div className={classes.buttonContainer}>
71            <Delete deleteEndpoint={`/api/research/${id}`} />
72            <Link
73              to={`/research/edit/${id}`}
74              style={{ textDecoration: "none", color: "inherit" }}
75            >
76              <Button
77                variant="outlined"
78                startIcon={<EditIcon />}
79                size="small"
80                onClick={() => {
81                  console.log("edit");
82                }}
83              >
84                Edit
85              </Button>
86            </Link>
87          </div>
88        )}
89        <Link
90          to={`/details/${id}`}
91          style={{ textDecoration: "none", color: "inherit" }}
92          className={classes.container}
93        >
94          <img src={img} alt="wrapper" className={classes.image} />
95          <Typography className={classes.heading}>{title}</Typography>
96          <Typography className={classes.text}>Author: {author}</Typography>
97          <Typography className={classes.text}>Category: {category}</Typography>
98          <LinesEllipsis
99            text={text}
100           maxLine="2"
101           ellipsis="..."
102           trimRight
103           basedOn="letters"
104           className={classes.detail}
105         />
106       </Link>
```

Figure 53: ResearchPost Component

## Server-Side Interaction

The back-end functionality was made using Node.js and a MongoDB database. The project utilizes the client-service architecture design. The client sends HTTP requests to specific endpoints that the back-end server understands in order to modify and access data. Certain HTTP requests require specific parameters to be sent along with the request, otherwise, the request will fail.

### Server Startup

The file that is in charge of server startup is server.js shown in Figure 54. One can locally start the server by opening a terminal window in the server directory and running *npm run dev*. On back-end server startup, the server first initializes an Express instance and stores it in the variable *app*. Express is a library that bootstraps a portion of the boilerplate code associated with creating a Node server. The server then attempts to connect to the project's MongoDB instance. This is implemented in a connectDB() function. The function utilizes Mongoose, a library for modeling JSON objects to MongoDB objects, which has a pre-built function to connect to a MongoDB instance by passing the instance's URI. One obtains the URI by retrieving it from the Wildlife Diseases MongoDB cluster; however, for convenience, the URI can be found in the default.json file located in the config directory.

After connecting to the MongoDB instance, the server initializes middleware that ensures incoming request payloads are of type JSON and not String. Afterward, the server defines its route endpoints. The server defines routes by declaring the base endpoint and then routing it to a separate file that contains the logic for all the endpoints associated with that service. For example, the server declares that all requests to endpoint */api/users* will be handled by logic contained in *./routes/api/users*. The final step the server takes to complete startup is to bind and listen to connections on a specific host and port. For local development, this defaults to localhost and port 5000. Now, the server will listen for any requests made to the specified host and port and perform specific logic depending on the endpoint of the request.

```
const app = express();

// Connect database
connectDb();

// Init Middleware
app.use(express.json( { extended: false }));
app.use(bodyParser.json());

app.get('/', (req, res) => {
  res.send('Hi there');
});

// Define Routes
app.use('/api/users', require('./routes/api/users'));   // makes /api/users pertain to '/' in users.js
app.use('/api/auth', require('./routes/api/auth'));
app.use('/api/research', require('./routes/api/researchPosts'));
app.use('/api/publication', require('./routes/api/publicationPosts'));
app.use('/api/about', require('./routes/api/aboutPosts'));

const PORT = process.env.PORT || 5000;

app.listen(PORT, () => {
  console.log('Listening on port ' + PORT);
});
```

Figure 54: Server.js Server Startup

## Data Models

The back end contains a number of Mongoose data models which translate directly to and define the fields associated with each of the MongoDB collections. The back-end server has four data models, and thus the MongoDB instance has four collections: ResearchPost, PublicationPost, AboutPost, and User. These four models deal with the four main forms of data relevant to the application. The client accesses each data model by sending requests to the appropriate endpoint. The files that define the four data models are located in the /models directory.

The User model corresponds to the data for logged-in system administrators. The User model consists of four fields: name, email, password, and date (see Figure 55). The name, email, and password are mandatory fields while the date of User creation is automatically set when the User instance is created. Additionally, the email field must be unique across the entire User collection. No two User instances may have the same associated email address.

```
const mongoose = require('mongoose');

const UserSchema = new mongoose.Schema({
    name: {
        type: String,
        required: true
    },
    email: {
        type: String,
        required: true,
        unique: true
    },
    password: {
        type: String,
        required: true
    },
    date: {
        type: Date,
        default: Date.now
    }
});
```

Figure 55: User Data Model

The AboutPost model shown in Figure 56 corresponds to the biography data of each member of the Wildlife Diseases Laboratory. The AboutPost model consists of eight fields: user, email, name, role, description, img, resumeLink, and date. The user field is not a biographical detail. Rather, it is the unique MongoDB ID associated with the User data model instance that created the AboutPost instance. The name, role, description, and img fields are mandatory fields. The role field refers to the laboratory member's status in the laboratory. For example, there are Professor, Graduate Student Researcher, Undergraduate Student Researcher, and Visiting Scholar roles. The description field is a brief narrative of the laboratory member while the img field is the URL of an image of the laboratory member stored in an AWS S3 bucket. The resumeLink field is non-empty if the AboutPost instance consists of a Professor role. The date of AboutPost creation is automatically set when the AboutPost instance is created.

```
const AboutPostSchema = new Schema({
  user: {
    type: Schema.Types.ObjectId,
    ref: "user",
  },
  email: {
    type: String,
  },
  name: {
    type: String,
    required: true,
  },
  role: {
    type: String,
    required: true,
  },
  description: {
    type: String,
    required: true,
  },
  img: {
    type: String,
    required: true,
  },
  resumeLink: {
    type: String
  },
  date: {
    type: Date,
    default: Date.now,
  },
});
```

Figure 56: AboutPost Data Model

The ResearchPost model shown in Figure 57 corresponds to the research post articles posted to the website. The ResearchPost model consists of eight fields: user, title, author, text, img, category, link, and date. Again, the user field is the unique MongoDB ID associated with the User data model instance that created the ResearchPost instance. The title, author, text, category, and link fields are mandatory fields. The category field helps separate the ResearchPost instances by the content of the posts. The text field is the body of the research post while the img field is the URL to a relevant image stored in an AWS S3 bucket. The link field is a link to an external article that directly corresponds to the contents of the ResearchPost instance.

```
const ResearchPostSchema = new Schema({
  user: {
    type: Schema.Types.ObjectId,
    ref: "user"
  },
  title: {
    type: String,
    required: true
  },
  author: {
    type: String,
    required: true
  },
  text: {
    type: String,
    required: true
  },
  img: {
    type: String,
  },
  category: {
    type: String,
    required: true
  },
  link: {
    type: String,
    required: true
  },
  date: {
    type: Date,
    default: Date.now
  }
});
```

Figure 57: ResearchPost Data Model

The PublicationPost model shown in Figure 58 corresponds to Professor Esobar's published works displayed on the website. The PublicationPost model consists of seven fields: user, title, author, publisher, year, link, and date. Again, the user field is the unique MongoDB ID associated with the User data model instance that created the PublicationPost instance. The title, author, publisher, and year fields are mandatory fields. The year field helps separate the PublicationPost instances by the age of the published works. The optional link field is a URL to the actual published work to which the PublicationPost instance correlates.

```
const PublicationPostSchema = new Schema({
  user: {
    type: Schema.Types.ObjectId,
    ref: "user"
  },
  title: {
    type: String,
    required: true
  },
  author: {
    type: String,
    required: true
  },
  publisher: {
    type: String,
    required: true
  },
  year: {
    type: String,
    required: true
  },
  link: {
    type: String,
    // required: true
  },
  date: {
    type: Date,
    default: Date.now
  }
});
```

Figure 58: PublicationPost Data Model

## Endpoints

The back-end server understands a number of endpoints: /auth, /researchPosts, /publicationPosts, and /aboutPosts. All routes that involve the updating of data, such as the creating and deleting of research posts, are protected routes - only a user with administrative privileges is allowed to access these endpoints. Postman is a software that creates HTTP requests given user-inputted routes, parameters, and payload. Postman was used to ensure endpoints would successfully execute without the need for client-side code to send the requests.

The auth endpoint contains all of the services related to authentication supported by the back end. The back end supports POST requests made to the auth endpoint. When the server receives a POST request to the auth endpoint, the request body is checked to ensure that the body contains an email and password. If not, the server would send an error response with an appropriate message, as seen in Figure 60, otherwise, a successful request is made as shown in Figure 59. If the two fields exist, the server will connect to the MongoDB instance and find if there is a corresponding email. If so, then the request body password field will be hashed using the project's secret key, and if this hashed password matches the password of the User instance

found to have the email of the request body email, the server attaches a JSON Web Token to the client.

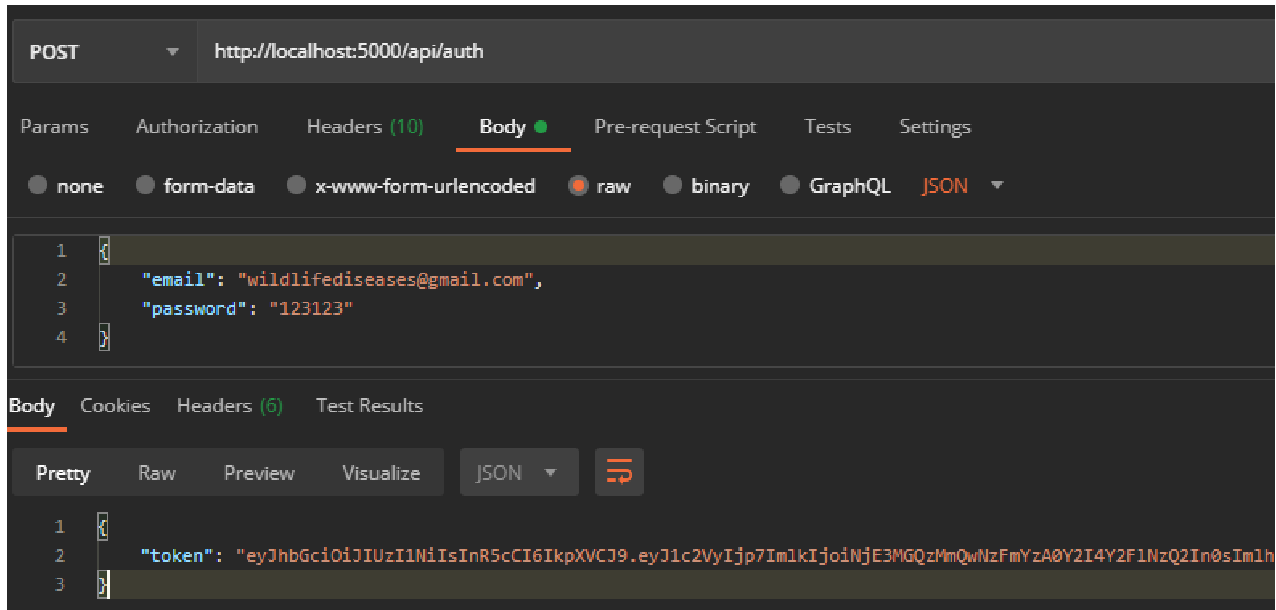



Figure 59: Successful POST Request to Auth Endpoint

There is no endpoint that allows for the creation of new users with administrator privileges as this could lead to security issues if a stranger learns of the endpoint, creates their own administrator user account, and is able to modify database collection entries.

When the backend server receives requests to protected endpoints, it will check for a valid JSON web token that is signed with the secret project signature. This valid JSON web token is only obtainable by the client via a response from the auth endpoint when the client sent a valid email and password combination.

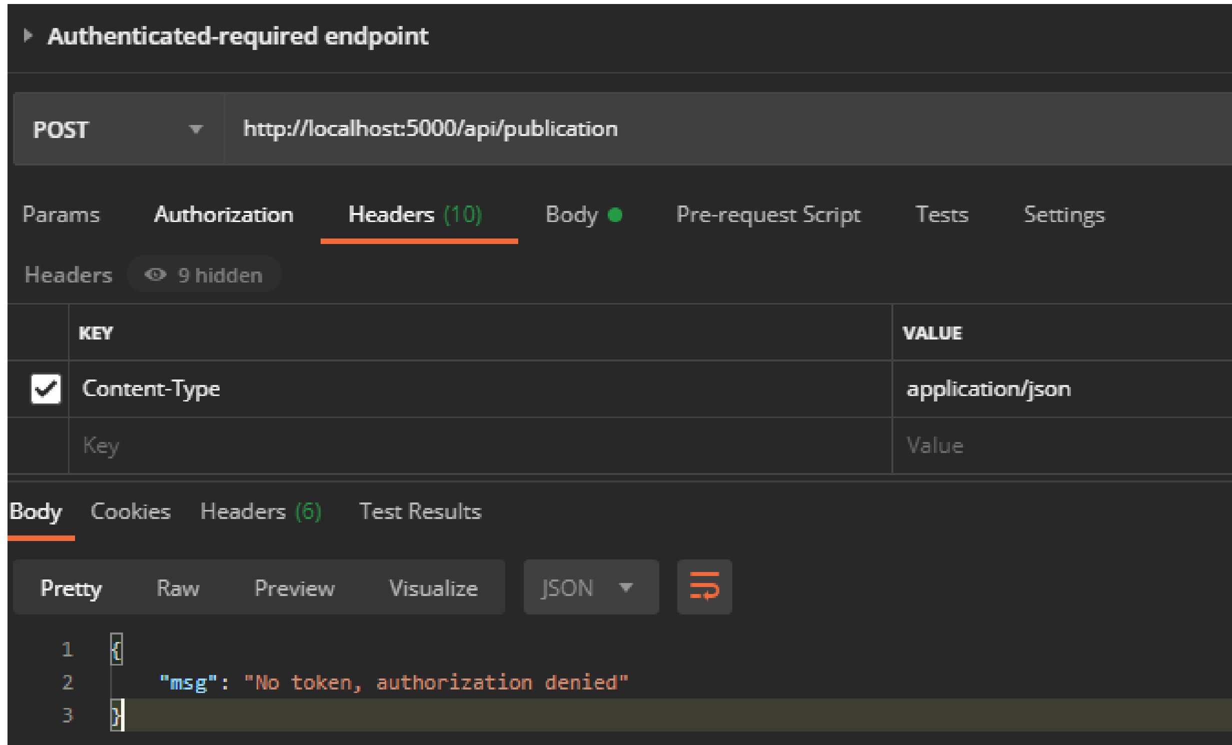



Figure 60: Authenticated Endpoints Require a JSON Web Token

The researchPosts endpoint holds all the services related to the research posts database collection - viewing, creating, and deleting research posts are all supported requests.

The endpoint that implements the retrieval of all research posts is a GET request to /api/research shown in Figure 61. When the server receives this GET request, it will not ensure there is a valid JSON Web Token attached as the endpoint is not an authenticated endpoint. The server uses the ResearchPost data model find() function with no filters in order to obtain all ResearchPost instances from the MongoDB instance. The server then sends the collection of ResearchPost instances as an array of JSON objects back to the client.

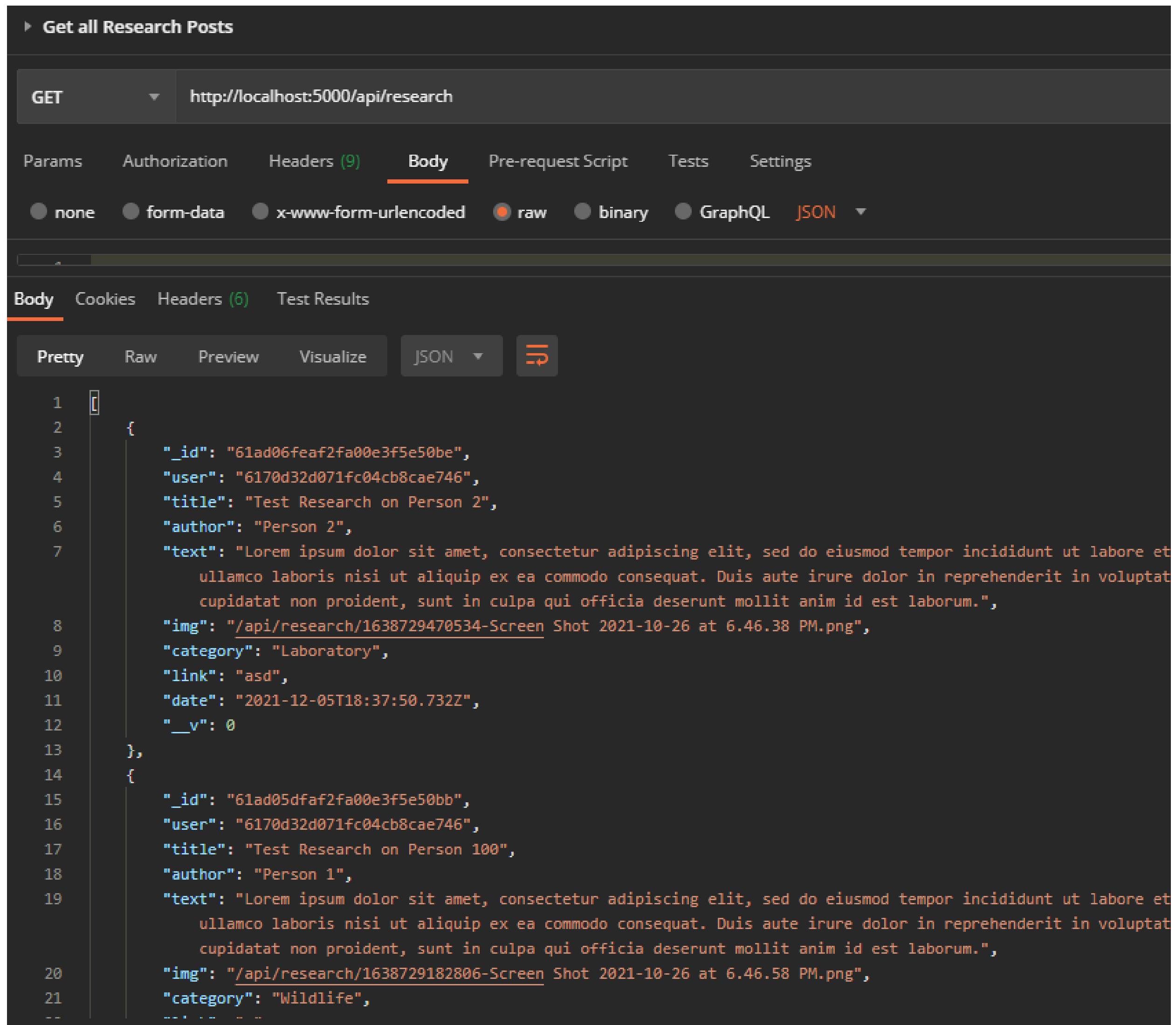



Figure 61: Retrieve all ResearchPost Instances

The endpoint that implements the creation of research posts is a POST request to /api/research shown in Figure 62. When the server receives this POST request, it will first ensure there is a valid JSON Web Token attached. Then, the server will locally upload the image file associated with the new research post. Once the file is locally uploaded, the server will upload the file to the project AWS S3 bucket. Once the file is successfully uploaded to the project AWS S3 bucket, the server deletes the local file. Then, the server creates a new ResearchPost data model instance using the fields in the request body. Finally, the MongoDB server saves the new ResearchPost instance, and the server sends the new ResearchPost instance JSON back to the client.

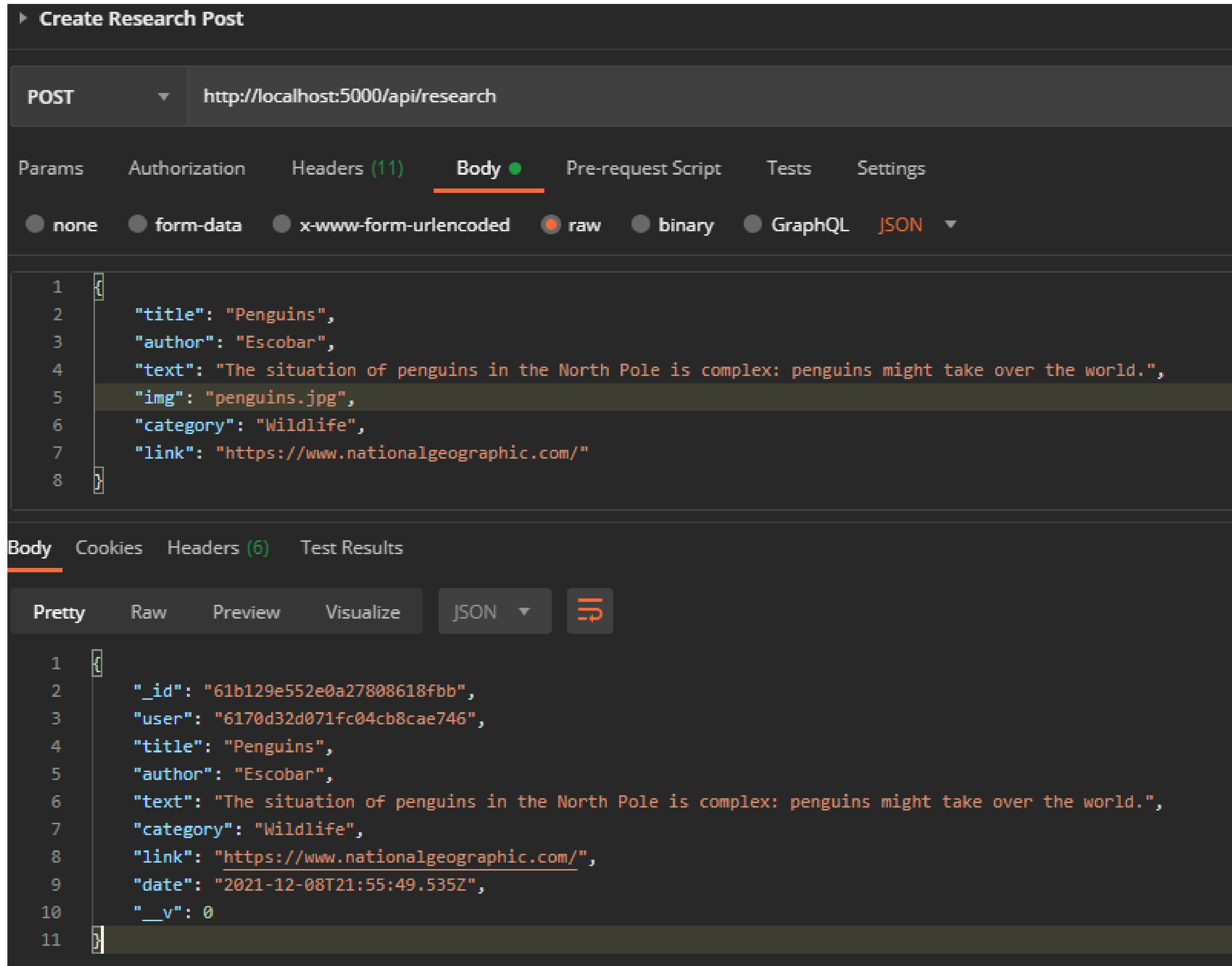



Figure 62: Successful ResearchPost Creation

The endpoint that implements the removal of research posts is a DELETE request to /api/research/:id. When the server receives this DELETE request, seen in Figure 63, it will first ensure there is a valid JSON Web Token attached. Then, the server will use the ResearchPost data model findById() function to locate the ResearchPost instance with the specified ID. The ID is passed to the endpoint as a parameter and the server can extract this parameter. If the server locates a ResearchPost instance, the server uses the instance's img field to obtain the URL of the stored image file in the AWS S3 bucket and deletes it from the S3 bucket. The MongoDB instance then deletes the instance from the database and the server attaches a "Post removed" message to the response.

Delete Research Post

DELETE http://localhost:5000/api/research/61b129e552e0a27808618fbb

Params Authorization Headers (11) Body Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL JSON

```
{
    "title": "Penguins",
    "author": "Escobar",
    "text": "The situation of penguins in the North Pole is complex: penguins might take over the world.",
    "img": "penguins.jpg",
    "category": "Wildlife",
    "link": "https://www.nationalgeographic.com/"
}
```

Body Cookies Headers (6) Test Results

Pretty Raw Preview Visualize JSON

```
{
    "msg": "Post removed"
}
```

Figure 63: Successful ResearchPost Removal

The publicationPosts endpoint holds all the services related to the publication posts database collection - viewing, creating, and deleting publication posts are all supported requests.

The endpoint that implements the retrieval of all publication posts is a GET request to /api/publication shown in Figure 64. When the server receives this GET request, it will not ensure there is a valid JSON Web Token attached as the endpoint is not an authenticated endpoint. The server uses the PublicationPost data model find() function with no filters in order to obtain all PublicationPost instances from the MongoDB instance. The server then sends the collection of PublicationPost instances as an array of JSON objects back to the client.

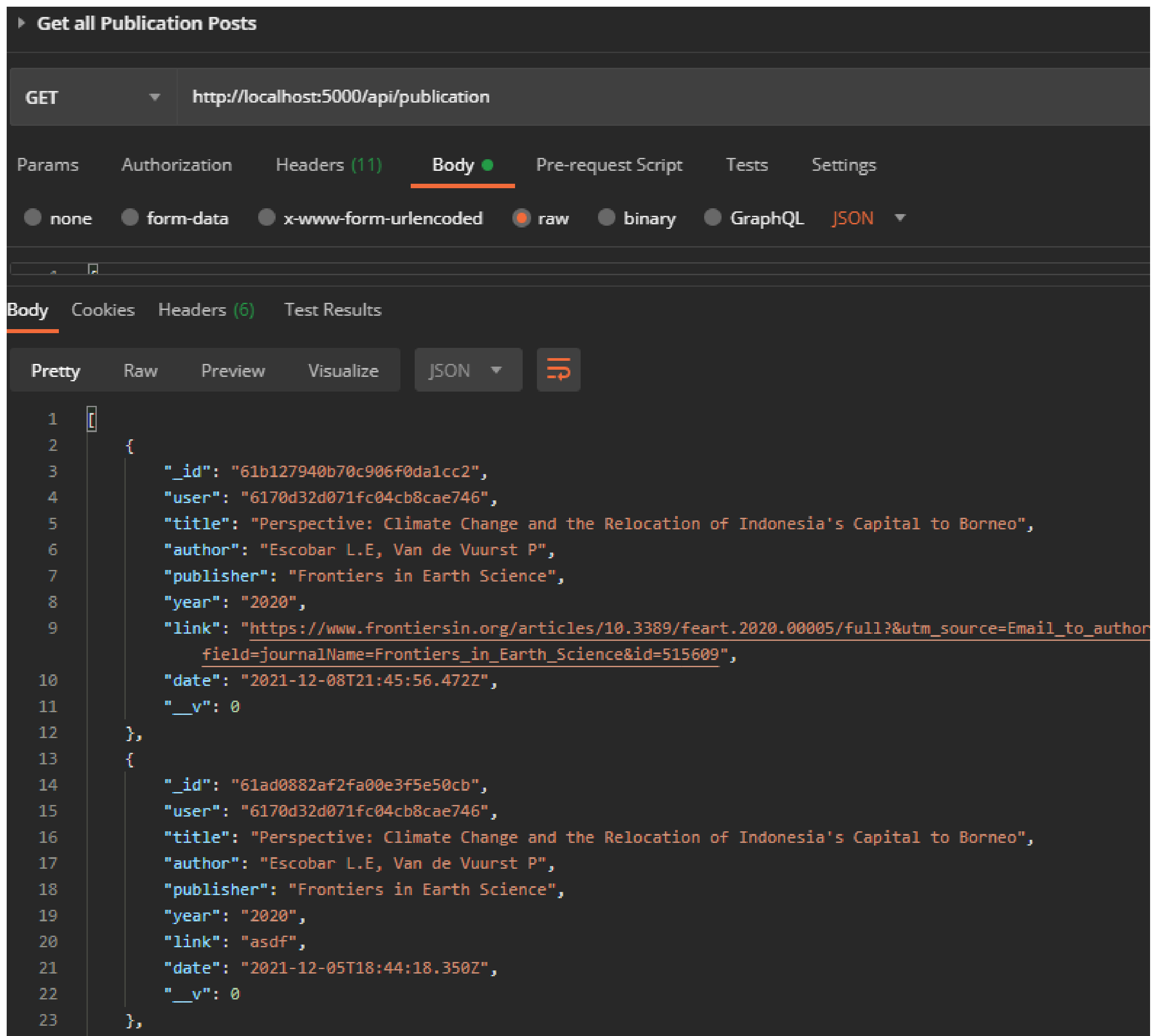



Figure 64: Retrieve all PublicationPost Instances

The endpoint that implements the creation of publication posts is a POST request to /api/publication shown in Figure 65. When the server receives this POST request, it will first ensure there is a valid JSON Web Token attached. The server creates a new PublicationPost data model instance using the fields in the request body. Finally, the MongoDB server saves the new PublicationPost instance, and the server sends the new PublicationPost instance JSON back to the client.

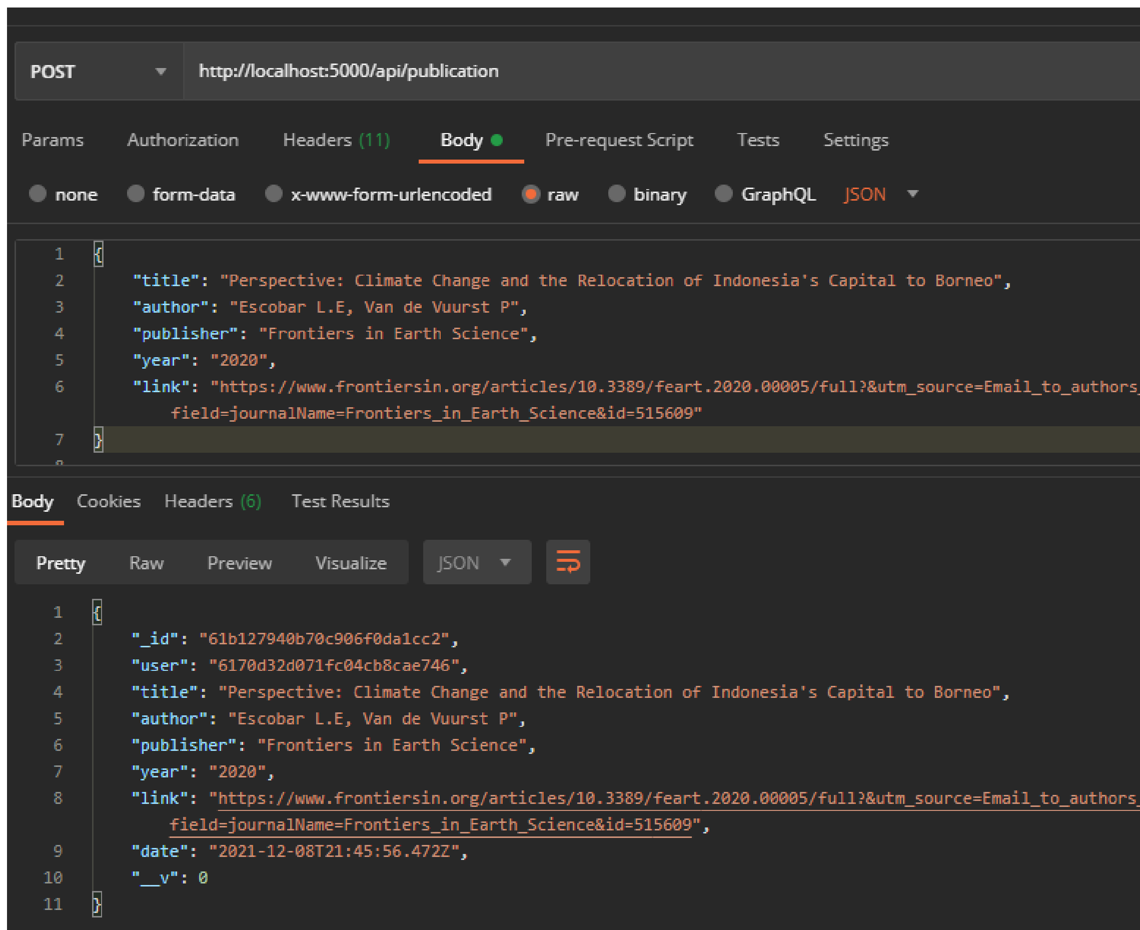



Figure 65: Successful PublicationPost Creation

The endpoint that implements the removal of publication posts is a DELETE request to /api/publication/:id shown in Figure 66. When the server receives this DELETE request, it will first ensure there is a valid JSON Web Token attached. Then, the server will use the PublicationPost data model findById() function to locate the PublicationPost instance with the specified ID. The ID is passed to the endpoint as a parameter and the server can extract this parameter. If the server locates a PublicationPost instance, the MongoDB instance deletes the instance from the database and the server attaches a “Post removed” message to the response.

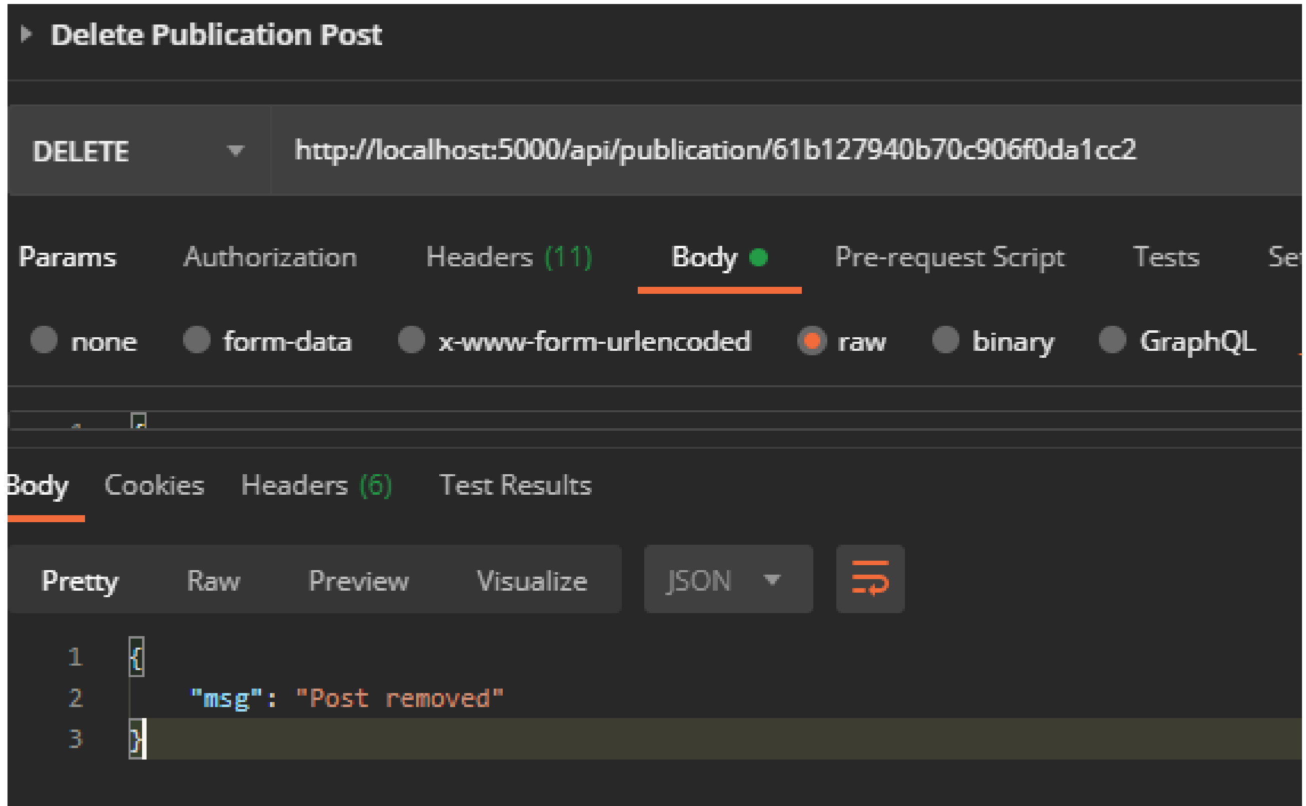



Figure 66: Successful PublicationPost Removal

The aboutPosts endpoint holds all of the services related to the lab students' biographies which are contained in the about posts database collection - viewing, creating, and deleting about posts and student biographies are all supported requests.

The endpoint that implements the retrieval of all publication posts is a GET request to /api/publication shown in Figure 67. When the server receives this GET request, it will not ensure there is a valid JSON Web Token attached as the endpoint is not an authenticated endpoint. The server uses the PublicationPost data model find() function with no filters in order to obtain all PublicationPost instances from the MongoDB instance. The server then sends the collection of PublicationPost instances as an array of JSON objects back to the client.

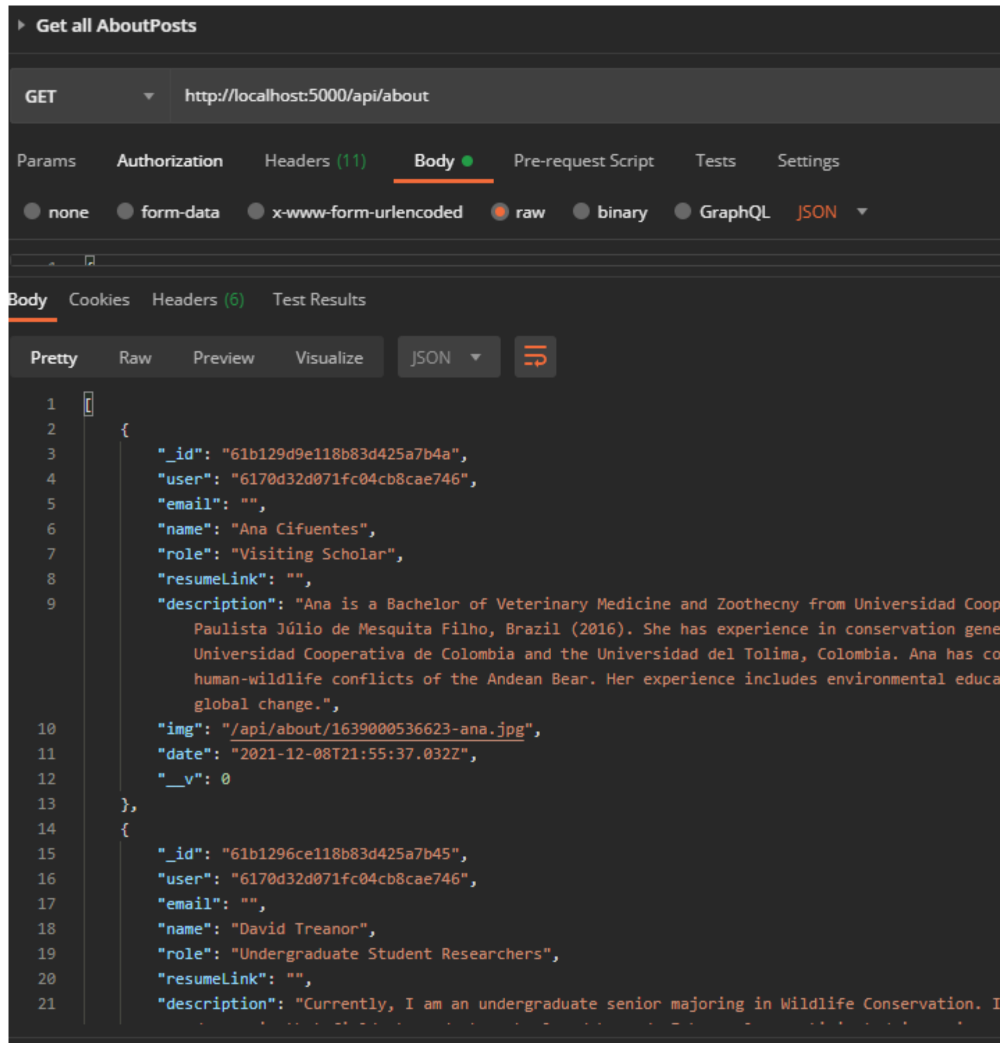



Figure 67: Retrieve all AboutPost Instances

The endpoint that implements the creation of about posts is a POST request to /api/about shown in Figure 68. When the server receives this POST request, it will first ensure there is a valid JSON Web Token attached. Then, the server will locally upload the image file associated with the new about post. Once the file is locally uploaded, the server will upload the file to the project AWS S3 bucket. Once the file is successfully uploaded to the project AWS S3 bucket, the server deletes the local file. Then, the server creates a new AboutPost data model instance using the fields in the request body. Finally, the MongoDB server saves the new AboutPost instance, and the server sends the new AboutPost instance JSON back to the client.

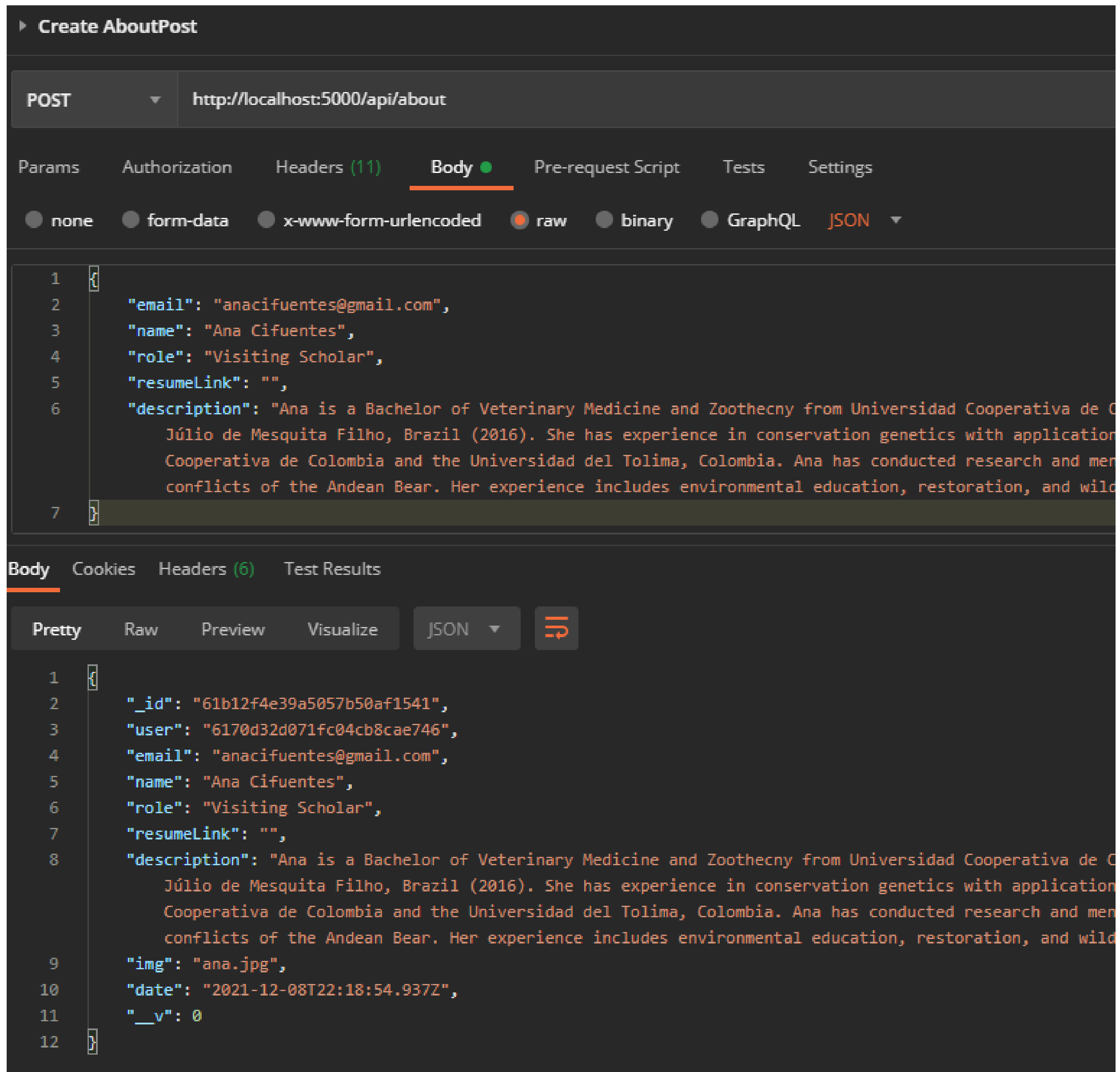



Figure 68: Successful AboutPost Creation

The endpoint that implements the removal of about posts is a DELETE request to /api/about/:id shown in Figure 69. When the server receives this DELETE request, it will first ensure there is a valid JSON Web Token attached. Then, the server will use the AboutPost data model findById() function to locate the AboutPost instance with the specified ID. The ID is passed to the endpoint as a parameter and the server can extract this parameter. If the server locates an AboutPost instance, the server uses the instance's img field to obtain the URL of the stored image file in the AWS S3 bucket and deletes it from the S3 bucket. The MongoDB instance then deletes the instance from the database and the server attaches a "Post removed" message to the response.

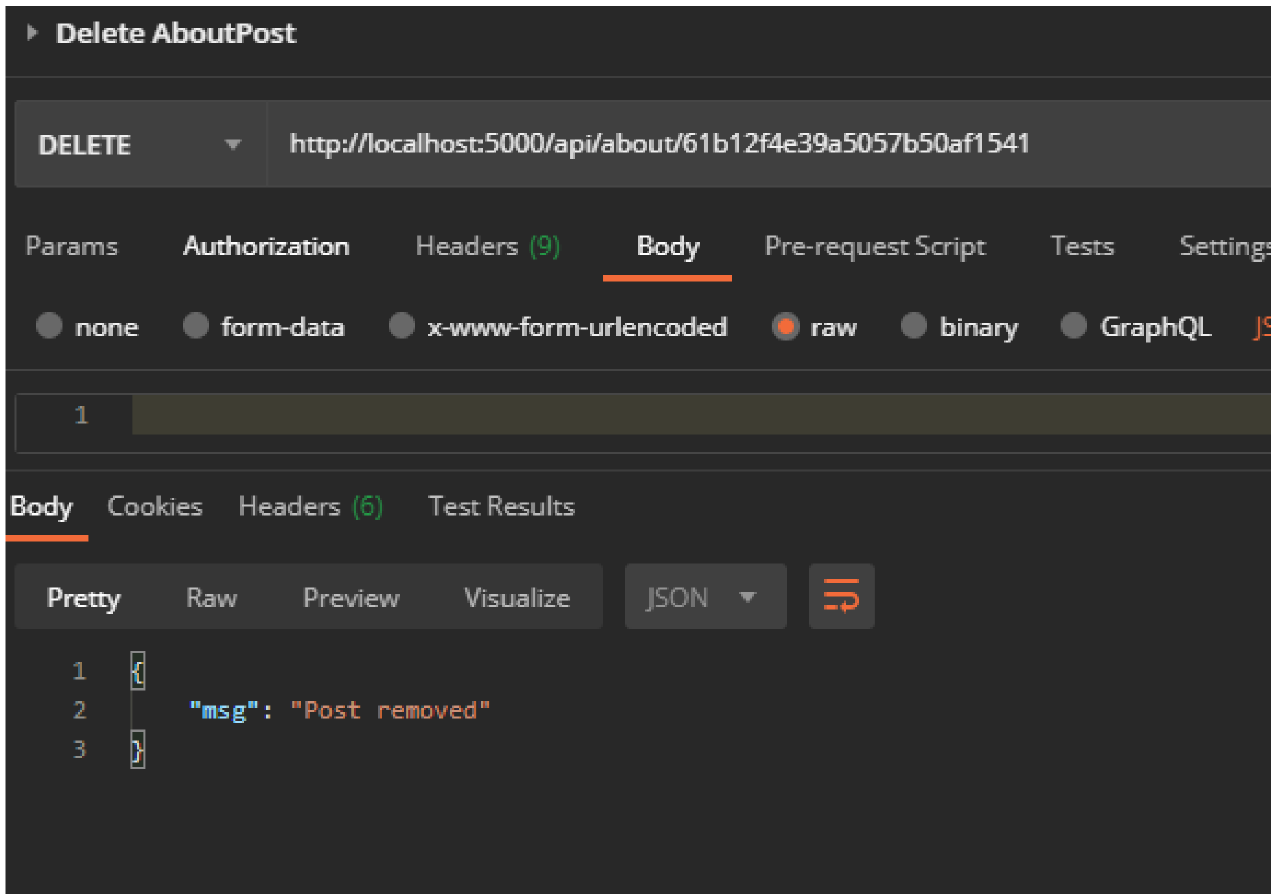



Figure 69: Successful AboutPost Removal

# Lessons Learned

The following section is a summary of the goals set by the team to complete the project as well as the challenges encountered when attempting to meet those deadlines.

## Timeline

Figure 70 describes the team's timeline including the goals for development and the dates of completion.

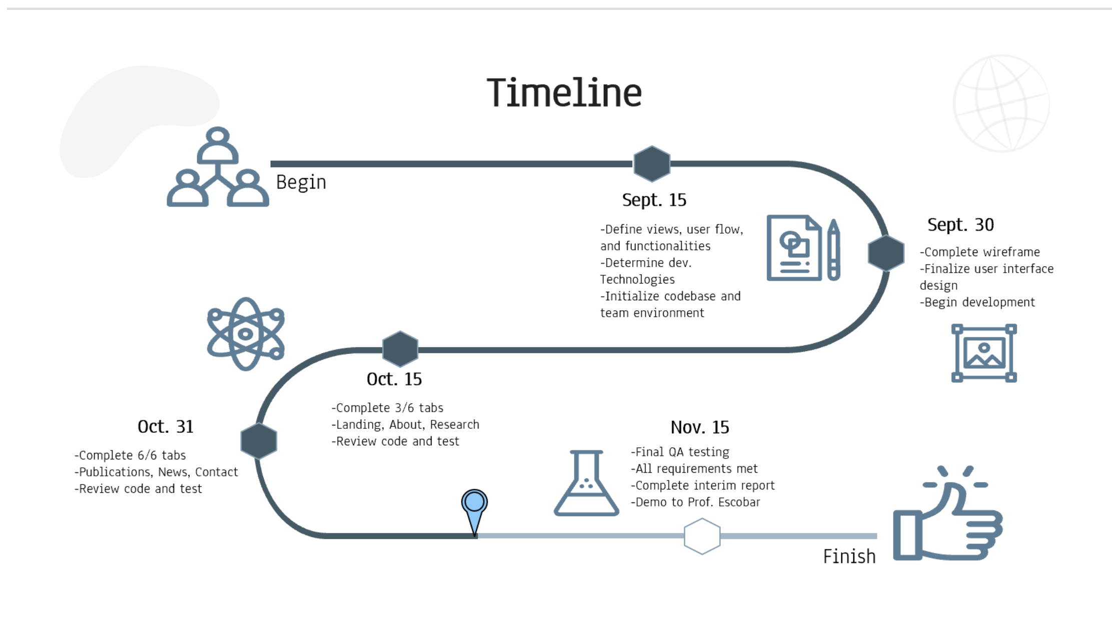



Figure 70: Timeline Chart

As shown in Figure 70, the timeline focuses on bi-weekly deliverables with each month being dedicated to a separate aspect of development. The month of September was dedicated to creating a front-end design meeting the specifications put forth by Professor Escobar. This goal was met on time and resulted in the creation of a UI design template and a set of wireframe sketches depicting the website's aesthetic and functionality.

The month of October was dedicated to the front and back-end development of each of the website pages. Three pages were to be fully designed every two weeks resulting in a complete and functional website by the end of October. This milestone was not fully completed by the desired due date; however, the team completed 70% of the required website functionality. The primary challenge was scheduling difficulties with other classes; however, there were no technical challenges associated with development. Another challenge was the issue of deployment. Finding a free hosting service for a dynamic website was a challenge, however, after discussions with Professor Fox and Escobar, a suitable solution was found.

## Deployment

Due to issues with hosting a dynamic website on free services, such as the VT hosting service, it was decided that the best solution was to host the website on a virtual machine. The virtual machine is defined as follows:

- Domain Name
  - wildlife.cs.vt.edu
- RAM
  - 8 GB of RAM
- Disk Space
  - 500 GB of Disk Space
- OS
  - The latest version of Ubuntu

After installing the necessary packages and libraries to generate the React Website, the website is hosted within the virtual machine using Nginx as long as the VM is available.

# References

[1] GeeksforGeeks, "Express.js: app.listen() Function." *GeeksforGeeks Website*, 6 Nov. 2021, retrieved Dec. 14, 2021, https://www.geeksforgeeks.org/express-js-app-listen-function/.

[2] Figma. (2021). *The Collaborative Interface Design Tool.* Figma. Retrieved December 14, 2021, from https://www.figma.com/.

[3] Microsoft. (2021, November 3). *Visual studio code - code editing. redefined*. RSS. Retrieved December 14, 2021, from https://code.visualstudio.com/.

[4] MongoDB Inc. (2021) *What is the MERN stack? Introduction & examples*. MongoDB. Retrieved December 8, 2021, from https://www.mongodb.com/mern-stack.

[5] W3C. (2013). *Markup Validation Service*. W3C.org. Retrieved December 14, 2021, from https://validator.w3.org/.

[6] Facebook Open Source. (2021). *React – A JavaScript library for building user interfaces*. Facebook, Inc. Retrieved December 8, 2021, from https://reactjs.org/.

# Acknowledgments

The team would like to thank Professor Escobar for the opportunity to assist him and his laboratory. This was a unique experience for the entire team. Professor Escobar can be contacted at escobar1@vt.edu. The team would also like to thank Professor Fox for his mentorship during this process. He can be contacted at fox@vt.edu.