Wildlife Tracker

Chase Hedelius, Marko Milenkovic, Sriteja Nara, Edris Shahem, Sean Johnson, and

Macon Bauknight.

Team: JIC 4125

Client: Jason Alstad, Wildlife Movement Institute

Repository: https://github.com/The-Enforcers/WildLife-Tracker-Repo-JIC4125

Table of Contents:

Table of Figures

Terminology

API (Application Program Interface) - A connection between two applications that allow for interaction between each other.

Backend - The side of the application that is typically only observed by those who are running the website or service.

Database - A location where a collection of information is stored, often connected to the backend.

Frontend - The side of the application that is able to be seen and typically interacted with by the user.

Controller - A backend component that manages the flow of data between the user interface (frontend) and the database. It processes user requests, interacts with models, and sends the appropriate response.

Model - Represents the data structure of a specific entity (e.g., User, Post, or Image) and defines the relationships between data stored in the database.

MongoDB - A NoSQL database used for storing application data in a flexible, JSON-like format.

React - A JavaScript library for building user interfaces, allowing developers to create reusable UI components for single-page applications.

Middleware - Software that acts as a bridge between the frontend and backend, often used in backend frameworks like Express.js to handle requests, perform authentication, and process data.

Authentication - The process of verifying the identity of a user or application, often through credentials such as usernames and passwords.

Authorization - The process of determining whether a user has permission to access specific data or perform specific actions.

CRUD Operations - The basic operations of persistent storage: Create, Read, Update, and Delete. These are fundamental to interacting with data in a database.

Introduction

Background

Our Project is Wildlife Tracker, a centralized web repository and interface that will allow for scientists and researchers to easily and quickly share information on wildlife trackers. The vast majority of information on wildlife trackers exists behind manufacturers' paywalls that prevent wide-access or are scattered on the web in researcher's papers, githubs, or blogs. Our platform would allow researchers to easily search for, access, and contribute information in regards to wildlife tracking.

Document Summary

The **System Architecture Section** provides a comprehensive overview of the structural design of the system, covering both static and dynamic aspects. The static design highlights the overall system components, their relationships, and dependencies, while the dynamic design focuses on the interactions between these components during specific use cases.

The **Component Design Section** dives deep into the specific functionalities of each feature within the project. This includes detailed descriptions of how components operate individually, their purpose within the system, and their contributions to fulfilling the overall project requirements.

The **Data Storage Design Section** explores the mechanisms used for managing the data that underpins the system. This includes a detailed explanation of how data for posts, user accounts, and tracker information is structured and stored. Additionally, this section elaborates on how data is sorted, retrieved, and managed to ensure efficiency, accuracy, and scalability.

The **UI Design Section** offers an overview of the website's user interface, providing insights into the user experience. It explains the layout, navigation, and aesthetic elements visible to the user, emphasizing how they contribute to intuitive interaction.

System Architecture

Introduction

To describe the structure of the Wildlife Tracking App, this document presents both static and dynamic architectural designs of the system. The main goals of the system are to: allow users to create posts about how to track specific animals, view others posts, edit/delete their own posts, like/bookmark posts, and search/filter through posts. The static system architecture diagram shows the different layers of the application, as well as how they depend on each other to accomplish the functions of the application. The dynamic system architecture diagram traces the flow of a common user task through the system, allowing us to better understand how the system works in real time.

Rationale

The design of the Wildlife Tracking App focuses on modularity, security, and usability. It is built as a full-stack application using the classic MERN stack. It employs React for a responsive frontend, Node.js + Express for the backend, and MongoDB for flexible storage. We use Google OAuth and JWTs (Json Web Tokens) to ensure secure session management. We have taken multiple security measures to safeguard against unauthorized access and abuse. These measures are: token-based checks to ensure users are properly authenticated, image upload limits to ensure our DB cannot be overloaded with massive images, and data rate limiting to prevent attackers from overwhelming our service with excessive traffic. Our backend is built modularly, split into controllers, routes, middleware, and utilities. Each component operates independently and performs specific functions, this makes the codebase much easier to maintain and expand upon. We chose to build this way to ensure future developers have an easy time extending and modifying our product to suit changing customer demands and needs.

Security Considerations -

Given that the Wildlife Tracker platform deals with potentially sensitive research data, security is a top priority. Authentication and authorization mechanisms are implemented to ensure that only authorized users can upload or modify posts. Furthermore, we employ industry-standard encryption protocols for data transmission between the frontend, backend, and database to prevent unauthorized access. Additionally, image uploads are validated and scanned to mitigate risks associated with malicious file uploads.

Static System Architecture

The static system architecture is driven by a full-stack application setup, where the frontend is built with React and the backend uses Node.js with Express and MongoDB as the database. This separation of concerns ensures that each component is specialized in its function, providing a clean architecture.

The application is divided into the following layers:

- **Frontend (Presentation Layer):** The React.js frontend is responsible for rendering the user interface (UI). It allows users to interact with the system by creating and viewing wildlife tracking posts. This interaction is made possible by the components and pages provided by React.
- **Backend (Business Layer):** The backend, built with Node.js and Express, handles the core logic of the application. It manages API endpoints that the frontend interacts with to send and retrieve data. The backend also integrates with MongoDB, a NoSQL database, to store wildlife tracking posts.
- **Database (Persistence Layer):** The MongoDB database stores the wildlife tracking posts and related information. It acts as the persistence layer that ensures data is safely stored and retrieved.

The interaction between the frontend, backend, and database is essential for maintaining the flow of data and ensuring that the system functions as expected. Below is a diagram representing this architecture.

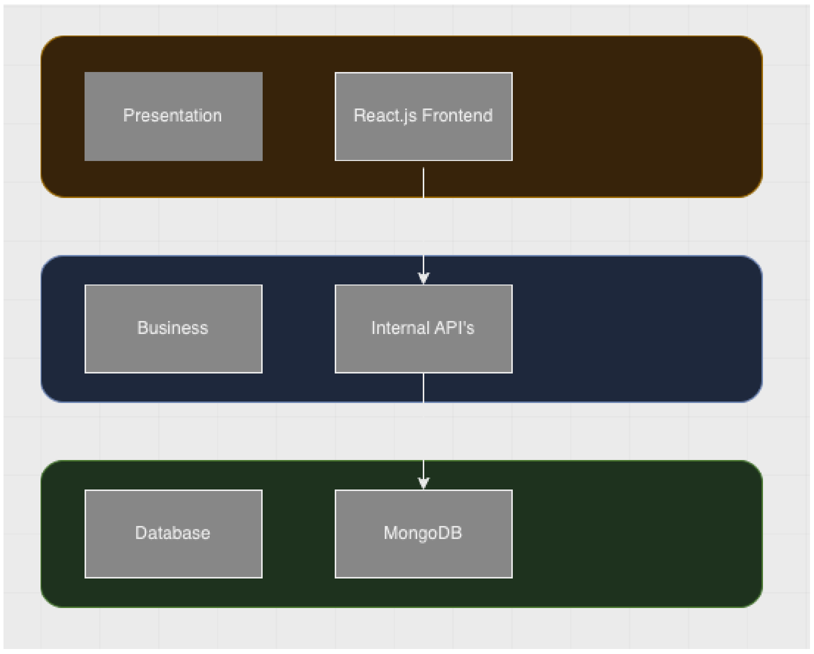


Figure 1 - Static System Diagram

Dynamic System Architecture

The system sequence diagram (SSD) below shows the flow of control between the parts of the Wildlife Tracking App as a user performs a common task. We chose to use a sequence diagram because we want to show the dynamic interactions between the systems components in chronological order, as we believe this will best illuminate the inner workings of our design. An SSD is perfect for this task. In this scenario we diagrammed, the user creates a new wildlife tracking post.

- **Step 1:** The user opens the application in the web browser
- **Step 2:** The user clicks the "Create New Post" button, which leads to a form where they can enter details about the wildlife sighting.
- **Step 3:** After filling out the form, the user submits it, and the frontend sends a POST request to the backend API to create the new post.
- **Step 4:** The backend receives the request and stores the new post data in the MongoDB database.
- **Step 5:** Once the post is saved, the backend sends a response back to the frontend, causing the frontend to show a success message

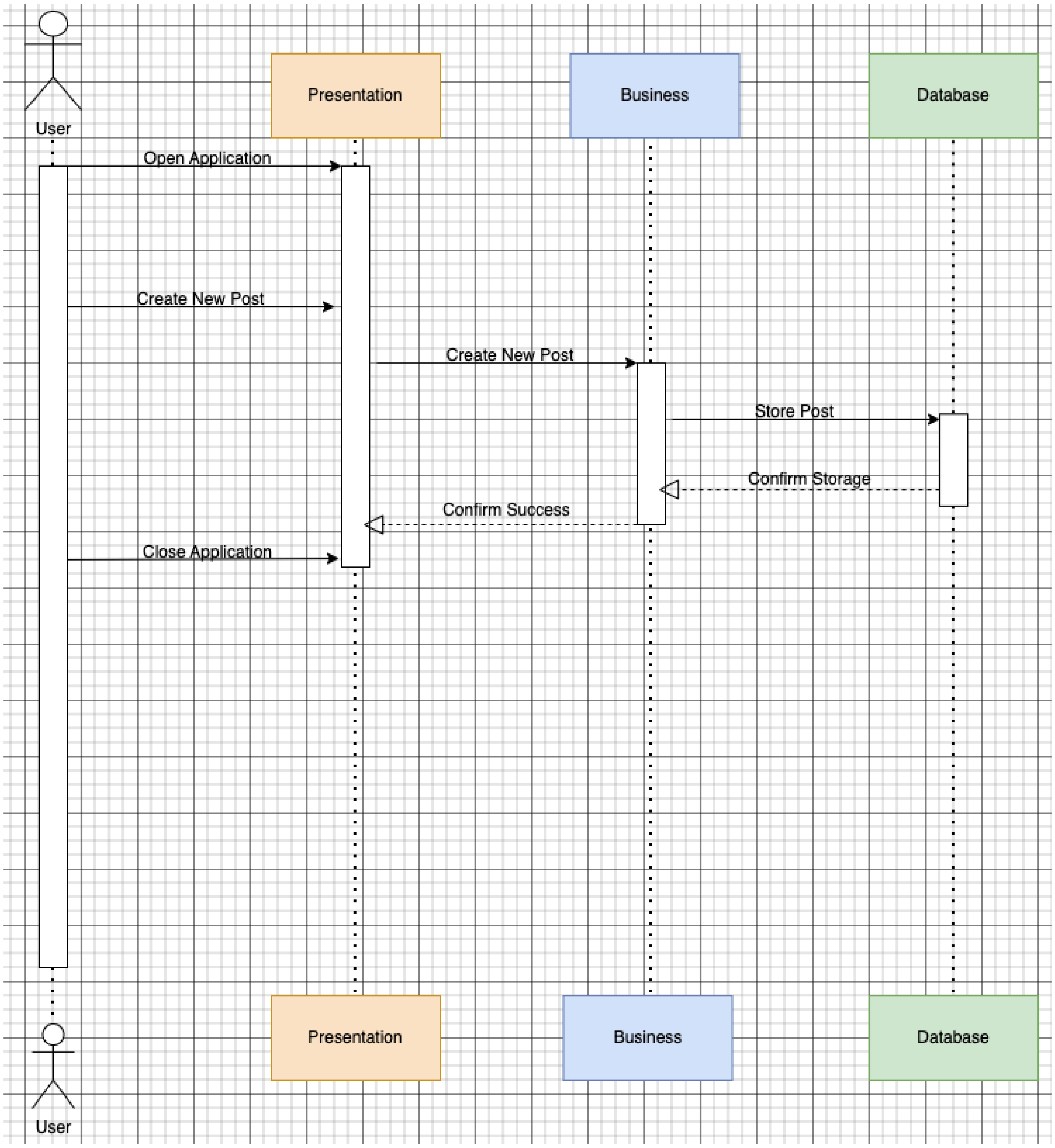


Figure 2 - Dynamic System Architecture

Component Design

Static Component Design

The static design of the Wildlife Tracker system is organized into three layers: **Frontend**, **Application Logic**, and **Data Layer**. The class diagram outlines the main components in each layer and their interactions.

1. **Web View Layer (Frontend)**:
   This layer consists of React components that form the user interface and enable interactions with the system. Each component is responsible for rendering specific elements of the application and communicating with services to update or retrieve data. The major components include:
   - **Main**: The primary page where users can view wildlife posts and interact with the search functionality. It maintains state variables and handles events like search and rendering posts.
   - **CreatePostPage**: This page allows users to create new posts by providing descriptions and uploading images. It interacts with the PostService to submit post data to the backend.
   - **PostDetailsPage**: Displays detailed information about a specific post. It retrieves post data from the PostService and renders the details to the user.
   - **ProfilePage**: Shows user-specific data and posts created by the logged-in user. It uses PostService to get posts related to the user and supports profile editing features.
   - **SearchBox**: A component that enables users to enter search terms and triggers search functionality. It sends search input to the Main component to filter displayed posts.
   - **Sidebar**: Provides navigation links to different pages and manages user navigation across the application.
2. **Application Logic Layer (Backend)**:
   This layer handles the core business logic of the application and is built using Node.js and Express. It manages requests from the frontend and interacts with the database. The main components include:

    - **PostController**: Manages CRUD operations related to posts, such as creating, updating, and deleting posts. It interacts with the Post model to access and manipulate data stored in the database.
    - **ImageController**: Handles image-related operations, such as uploading and retrieving images. It uses the Image model to store metadata and file paths in the database.
    - **PostRoutes**: Defines the API endpoints for handling requests related to posts and images. It connects routes to corresponding controller methods, such as createPost and uploadImage.
    - **PostService**: Located on the frontend, PostService is responsible for sending HTTP requests to backend endpoints to retrieve or modify data. It serves as an intermediary between frontend components and backend controllers.
    - **Server**: The central component of the backend, responsible for starting the server, configuring routes, and initializing middleware. It interacts with PostRoutes to ensure that API endpoints are correctly set up.
3. **Data Layer (Database)**:
   The data layer includes MongoDB models that define the structure of data stored in the database. These models interact with the controllers to perform data-related operations. The main components are:
    - **Post Model**: Represents a post in the system with attributes such as title, description, imageURL, and author. The model provides methods to create, update, or delete posts in the database.
    - **Image Model**: Represents image data with attributes such as fileName, filePath, and uploader. It allows for storing image metadata and retrieving or deleting images.
    - **User Model**: Represents user data, including attributes like username, email, and password. The model is used for creating, retrieving, or deleting user accounts.

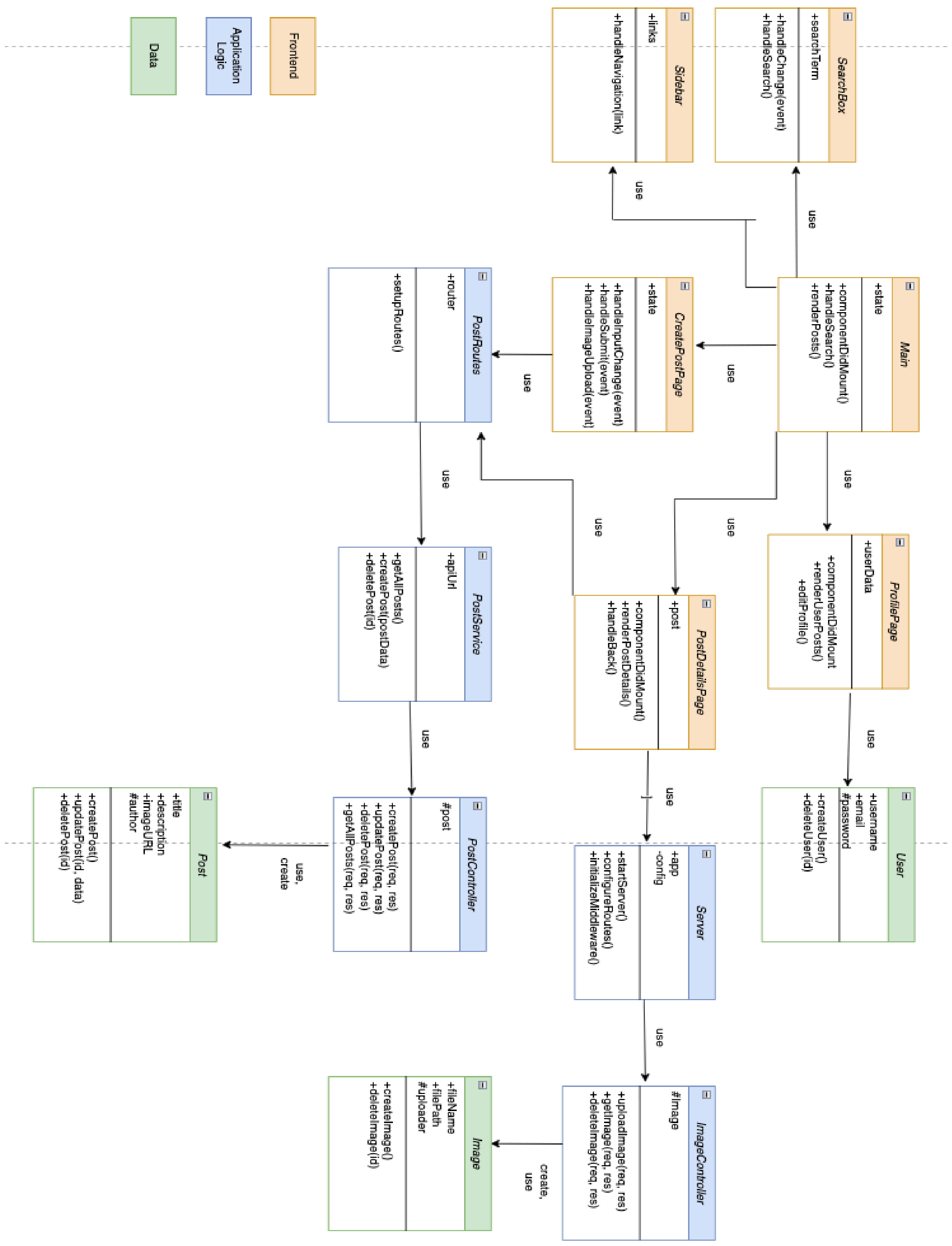


Figure 3 - Static Component Design

Dynamic Component Design

SearchPage Component:

- Contains:
  - A **Search Input** field for the user to enter keywords or criteria.
  - A **Search Button** to initiate the search operation.
  - **Filter Options** for refining search results based on specific attributes.
  - A **Results Display** area to show the results retrieved from the database.
- Interaction:
  - When the user inputs data and clicks the Search Button, the data is sent to the **API Gateway/Server**.
  - The API Gateway communicates with the **Database** to fetch relevant results based on the user's input.
  - The retrieved data is sent back to the SearchPage for display in the Results Display section.

CreatePostPage Component:

- Contains:
  - A **Post Title Input** field for entering the title of a post.
  - A **Post Description Input** field for providing a description.
  - An **Image Uploader** to upload related images.
  - A **Submit Button** to send the post data to the server.
- Interaction:
  - When the user fills in the inputs and uploads images, the data is sent to the **API Gateway/Server** upon clicking the Submit Button.
  - The API Gateway processes this data and stores it in the **Database**.
  - The post data is saved in the **Post Model**, and any associated images are saved in the **Image Model** within the database.

API Gateway/Server:

- Acts as the intermediary between the user-facing components (SearchPage and CreatePostPage) and the database.
- Handles requests:
  - From the SearchPage: Retrieves filtered or keyword-matched data from the Post Model or Image Model in the database.

    - From the CreatePostPage: Processes new posts and images, and saves them to the corresponding models in the database.

Database:

- Composed of two models:
    - **Post Model**: Stores textual data related to posts (e.g., titles and descriptions).
    - **Image Model**: Stores image files or references uploaded through the Image Uploader.

User Interaction:

- Users can either search for posts using the SearchPage or create new posts using the CreatePostPage.
- The API Gateway ensures efficient and secure communication between the user interface and the database for both retrieval and storage operations.

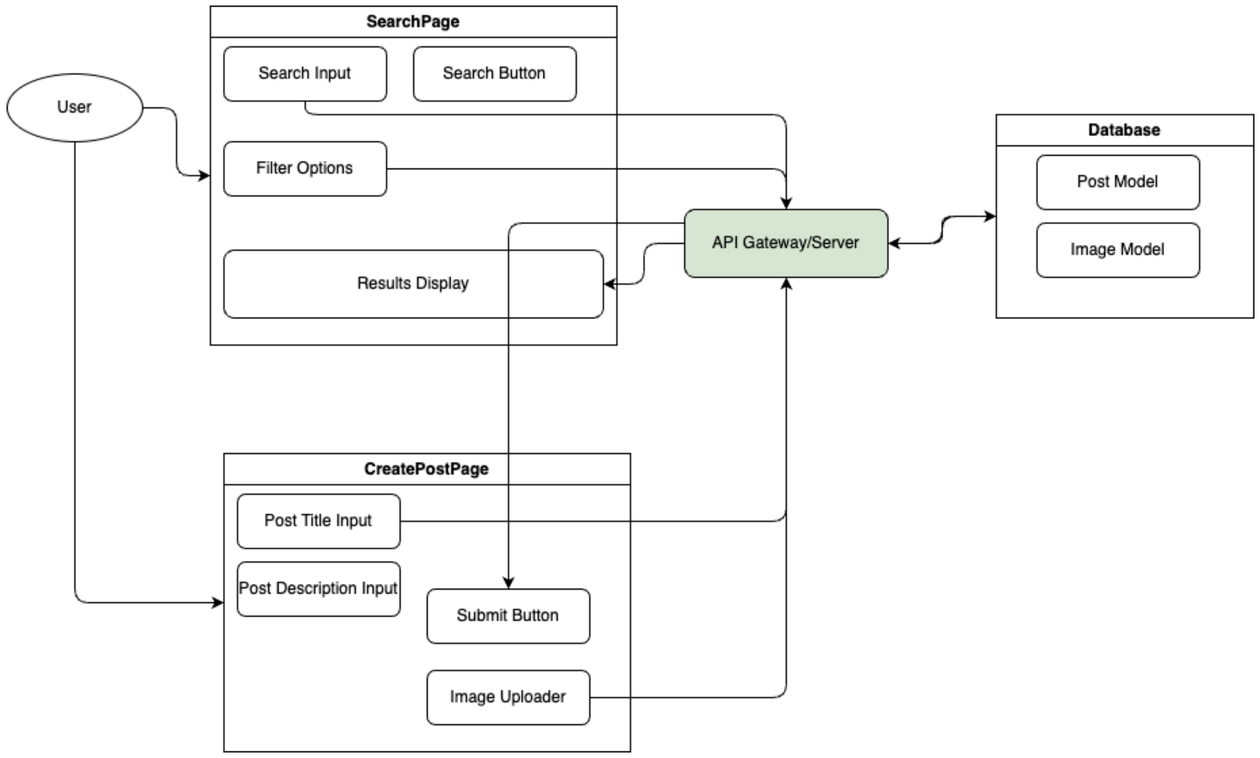


Figure 4 - Dynamic Component Design

Data Storage - MongoDB

Introduction

This section covers how we handle data, files, and security for the wildlife tracker repository. The website uses a **non-relational database (MongoDB)** to store data in a flexible, document-based format like JSON. For example, each record contains details about an animal, its tracker, and the researcher who posted it. This structure allows easy organization of data without the need for complex relationships between tables.

Files that we store

The site supports files like **images** (e.g., .jpg, .png) and **research documents** (e.g., .pdf), which are uploaded securely and stored in the cloud. For exchanging data between the website and users, we use **JSON** format over **HTTPS** to ensure all communications are secure.

Login and security

To ensure security, **Google OAuth** is used for user login, meaning passwords aren't stored, and user data is protected. Data encryption is applied at both the database level and during transfers to prevent unauthorized access. Our website uses **https** which means there is a certificate of validity. Sensitive information like researcher contact details is encrypted, and privacy is protected by allowing only authorized users to access or update specific data. Overall, we prioritize secure data handling and privacy throughout the platform.

Saving Process

The process of saving posts and images in MongoDB starts when a user interacts with the frontend of the application. The frontend presents a form for the user to submit a new post, which includes the post data (such as title and description) and an image.

Once the user submits the form, the PostService sends the post and image data to the BackendAPI. The API delegates these tasks to the appropriate controllers, with the PostController handling the post data and the ImageController managing the image.

Both the PostController and ImageController use the PostModel and ImageModel, respectively, to format the data for storage in MongoDB.

The PostModel stores information such as the post ID, title, description, and the associated image, while the ImageModel stores the image URL, image ID, and the related post ID. Once the data is saved in MongoDB, a confirmation is sent back through the layers to the frontend, where the user is notified of the successful post submission.

The **PostModel** is responsible for storing details such as the post ID, title, description, and its associated image.

Meanwhile, the **ImageModel** manages data like the image URL, image ID, and the corresponding post ID.

After the information is successfully saved in MongoDB, a confirmation is propagated back through the system layers to the frontend, where the user is informed of the successful submission.

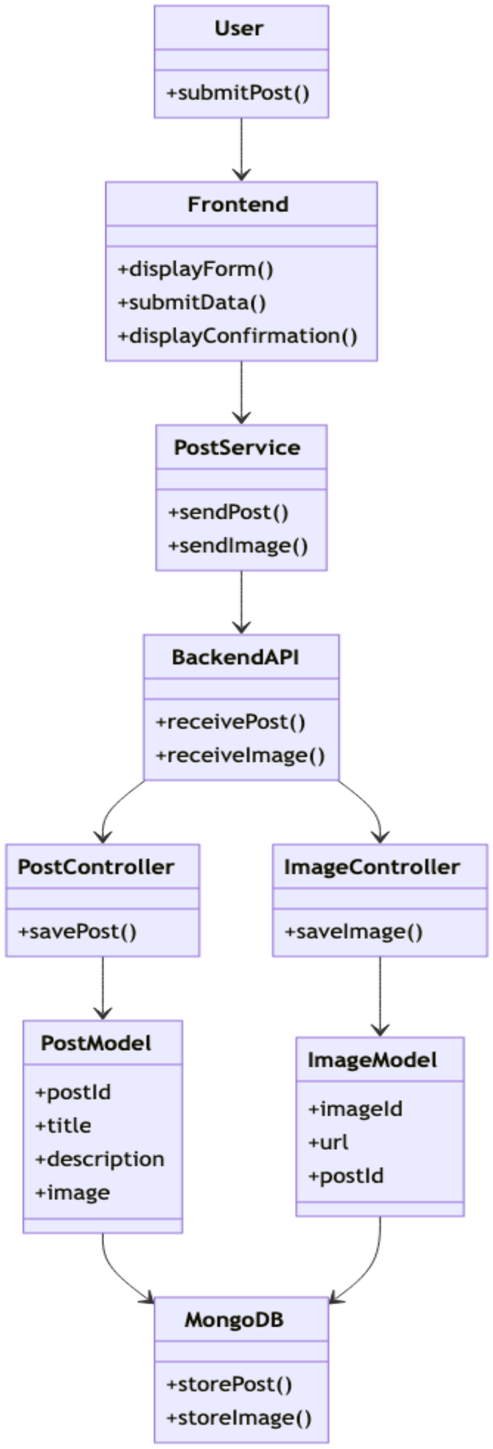


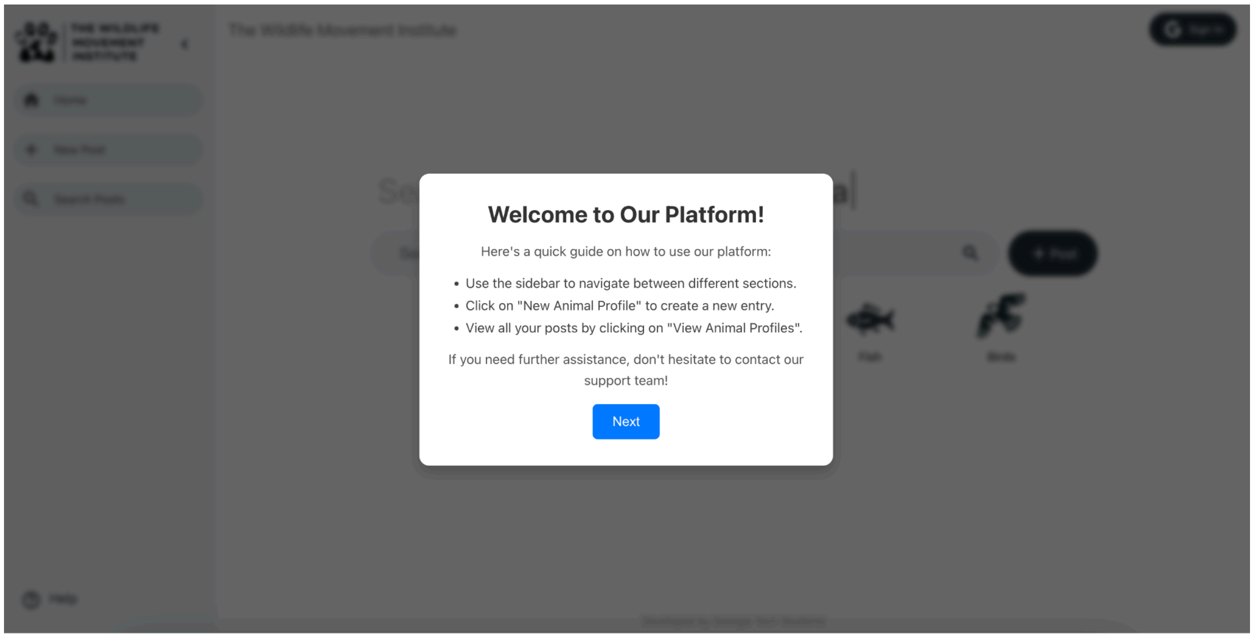
Figure 5 - Data Design

UI Design & Appendices

Introduction

The **Wildlife Tracker** project is a centralized platform where scientists and researchers can efficiently share and access information about animal trackers. This repository seeks to democratize access to data currently locked behind paywalls or scattered across various academic and online platforms. The application's core functions include creating and viewing posts about wildlife trackers, searching for specific trackers, and sharing posts with other researchers, therefore creating a centralized data repository of these animal trackers and how they should be used on different animals. Each screen is designed to support these functions while ensuring seamless navigation and interaction for users.

Welcome Screen - First time users will be seeing this welcome screen before using our website for the first time. After this a tutorial will appear showing users how to use our website.



Welcome to Our Platform!

Here's a quick guide on how to use our platform:

- Use the sidebar to navigate between different sections.
- Click on "New Animal Profile" to create a new entry.
- View all your posts by clicking on "View Animal Profiles".

If you need further assistance, don't hesitate to contact our support team!

Figure 6 - Welcome Screen

Main Screen - This is our homepage where users can login with their google account in order to post (sign in UI is handled by google). You can search through existing posts without having to login. This page also consists of quick links to search for specific species.

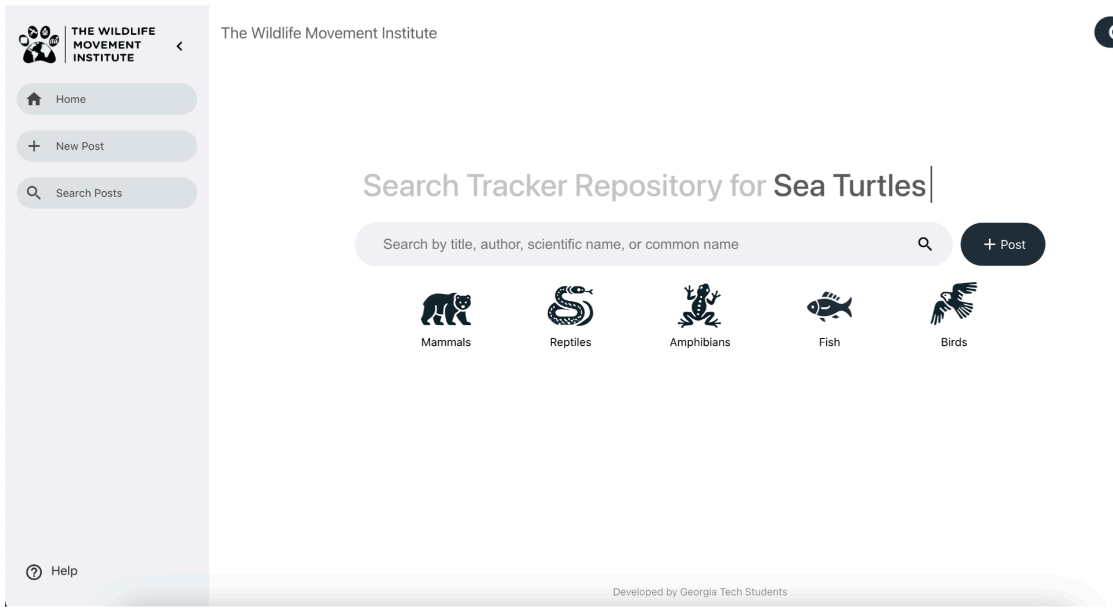
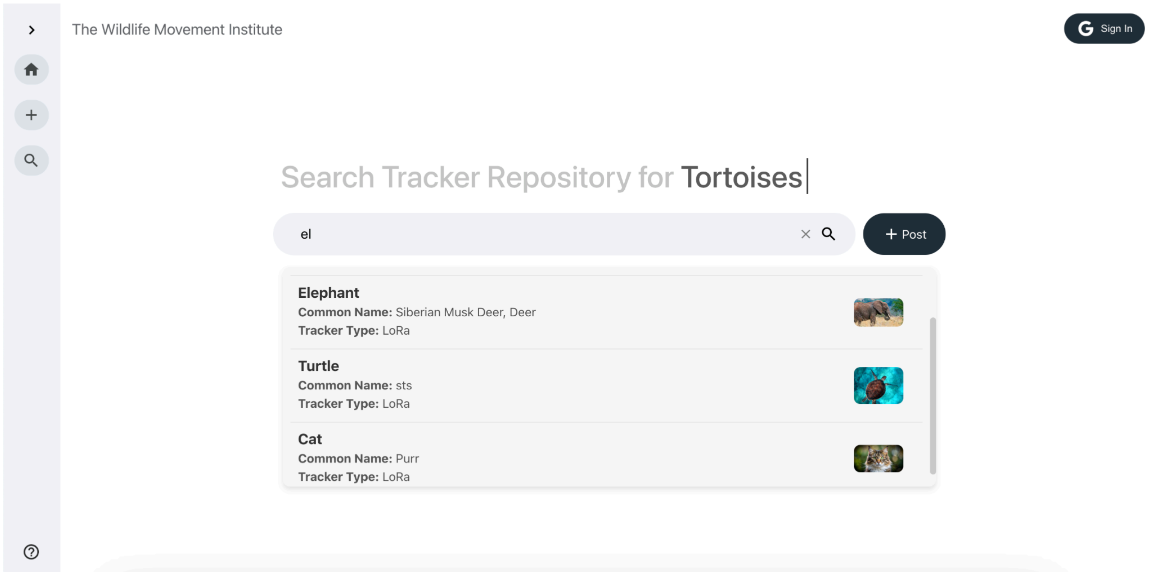

THE WILDLIFE MOVEMENT INSTITUTE
Home
New Post
Search Posts
Help
The Wildlife Movement Institute
Sign In
Search Tracker Repository for Sea Turtles
Search by title, author, scientific name, or common name
+ Post
Mammals
Reptiles
Amphibians
Fish
Birds
Developed by Georgia Tech Students

Figure 7 - Main Screen

Main screen with search- Displays a feed of wildlife tracker posts with search functionality for easy access to specific posts.

Sidebar Navigation - Facilitates movement across different sections of the application, including links to the Main, Create Post, and Search for posts. Consistent positioning and labeling of sidebar items ensure users can navigate intuitively, improving user flow.


The Wildlife Movement Institute

Sign In

Search Tracker Repository for Tortoises

el

Post

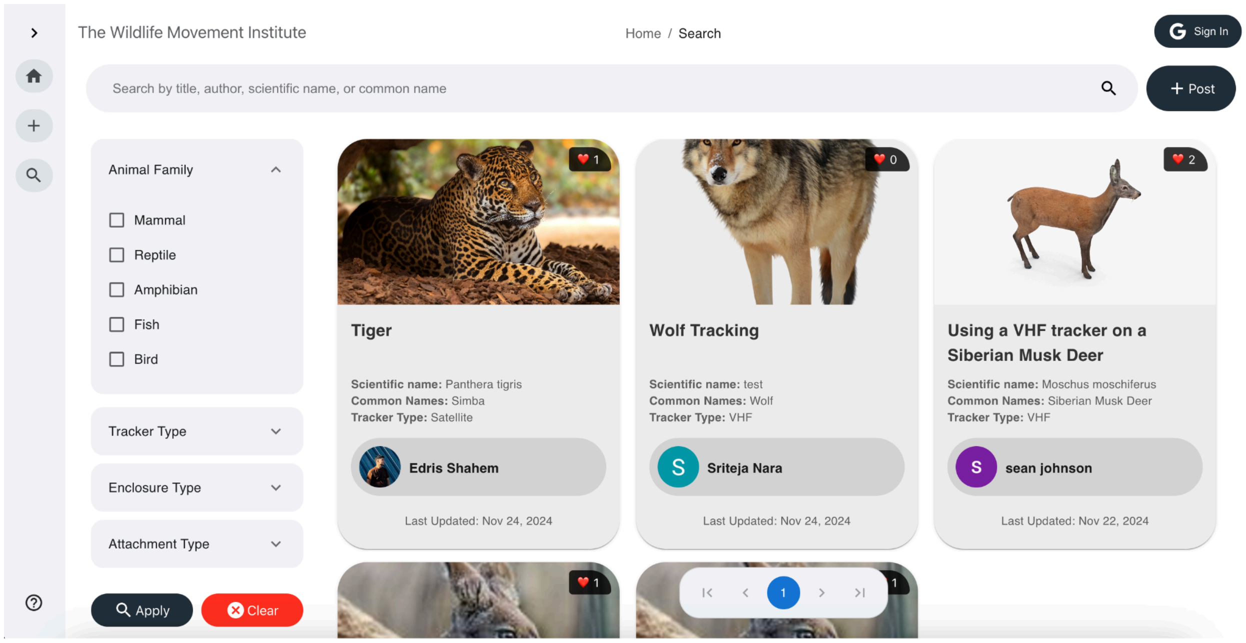
Elephant
Common Name: Siberian Musk Deer, Deer
Tracker Type: LoRa

Turtle
Common Name: sts
Tracker Type: LoRa

Cat
Common Name: Purr
Tracker Type: LoRa

Figure 8 - Main Screen with Search

Search Page with Adv Filters - Clear, search and filter options ensure users can quickly find relevant content, and a list of posts is visible with essential information like tracker name, type, and image preview.


The Wildlife Movement Institute

Home / Search

Sign In

Search by title, author, scientific name, or common name

Post

Animal Family
- Mammal
- Reptile
- Amphibian
- Fish
- Bird

Tracker Type

Enclosure Type

Attachment Type

Apply

Clear

Tiger ❤ 1
Scientific name: Panthera tigris
Common Names: Simba
Tracker Type: Satellite
Edris Shahem
Last Updated: Nov 24, 2024

Wolf Tracking ❤ 0
Scientific name: test
Common Names: Wolf
Tracker Type: VHF
Sriteja Nara
Last Updated: Nov 24, 2024

Using a VHF tracker on a Siberian Musk Deer ❤ 2
Scientific name: Moschus moschiferus
Common Names: Siberian Musk Deer
Tracker Type: VHF
sean johnson
Last Updated: Nov 22, 2024

Figure 9 - Search with Advanced Filters

Create Animal Profile page - Allows users to contribute information about new wildlife trackers by filling out form fields, including title, description, and image upload. Form validation provides immediate feedback on submission requirements, and each element follows a logical, easy-to-follow layout.

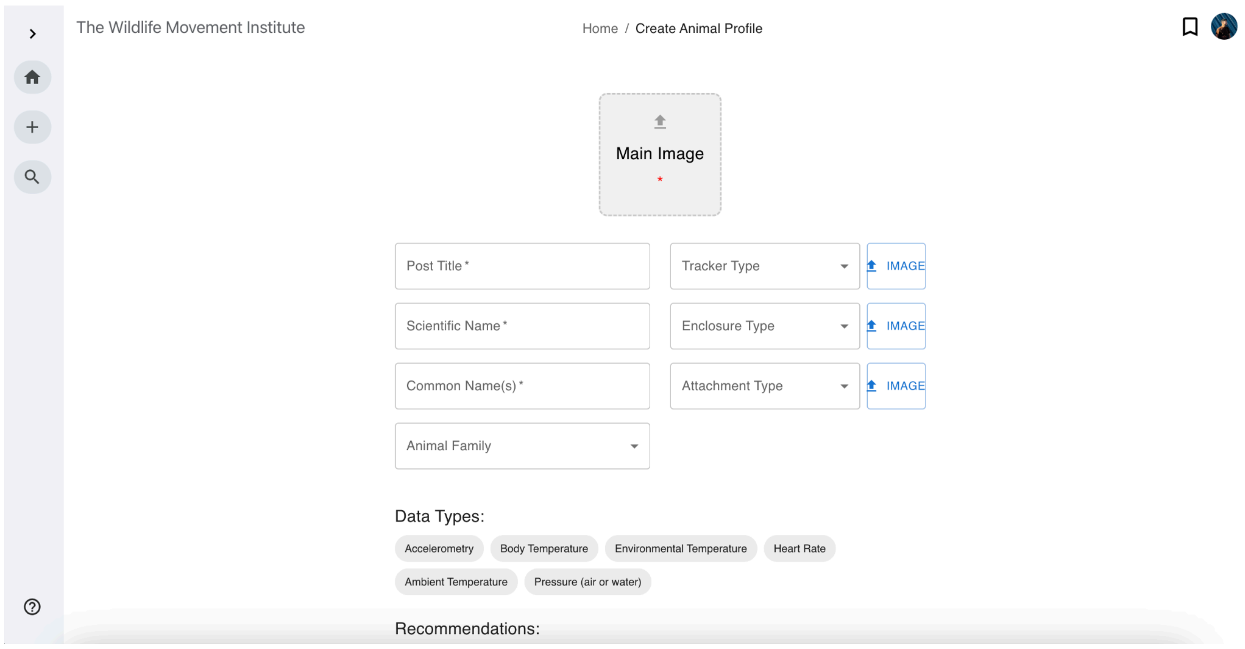


Figure 10 - Create Animal Profile Page

Post Details Page - Shows a detailed view of a selected wildlife tracker, with information on the tracker's features, type, and user-uploaded images. Navigation options are consistently placed, allowing users to return to the main feed or search for other trackers.
Bookmarking and liking the posts are also some features of this page. You can view bookmarked posts by clicking the icon on the left of the profile picture.

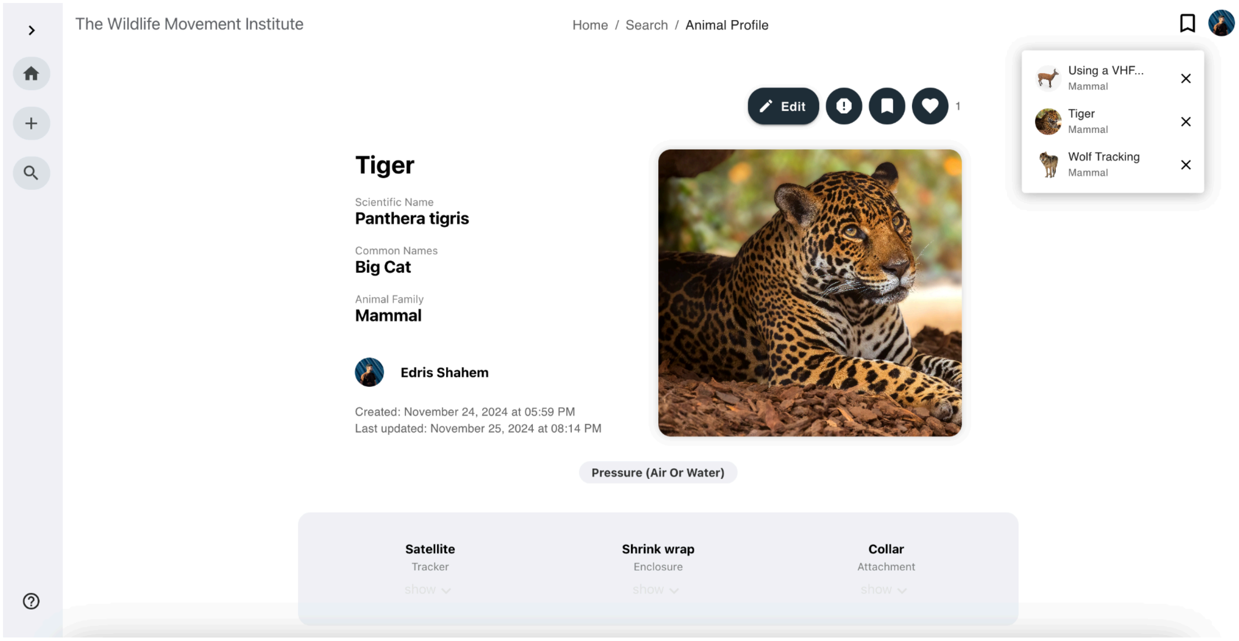


Figure 11 - Post Details Page

Profile Page - Displays user-specific data, including the number of contributed posts and profile information. Users can edit their profiles, manage their posts, and view their recent posts within an organized and accessible layout.

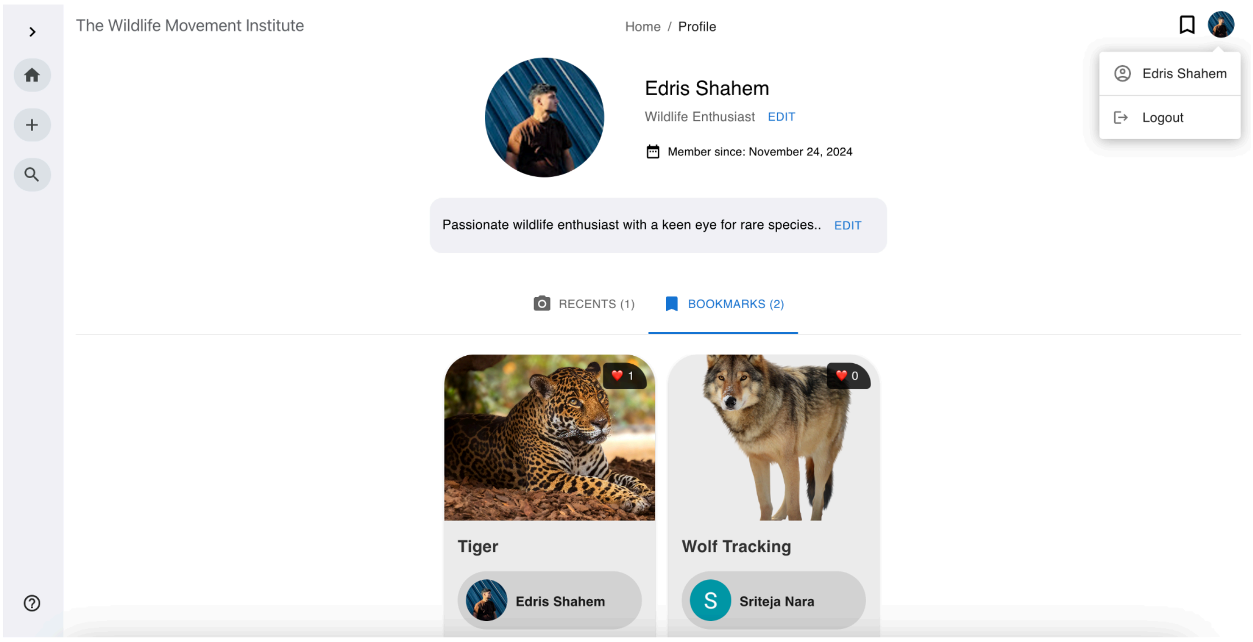


Figure 12 - Profile Page

Appendix

Google OAuth API

Resource URL

https://developers.google.com/identity/protocols/oauth2

Resource Information

- **Response Format:** JSON
- **Requires Authentication:** Yes
- **Rate Limited:** Based on Google Cloud quotas

Parameters

- Client ID
- Client Secret
- Redirect URI
- Scope
- State (optional)

Example Request

```
POST https://oauth2.googleapis.com/token

Content-Type: application/x-www-form-urlencoded

client_id=YOUR_CLIENT_ID&

client_secret=YOUR_CLIENT_SECRET&
```

```
code=AUTHORIZATION_CODE&

grant_type=authorization_code&

redirect_uri=YOUR_REDIRECT_URI
```

Example Response

```
{
  "access_token": "ya29.a0ARrdaM9...5HgQ",
  "expires_in": 3599,
  "refresh_token": "1//0gfuR...sw",
  "scope": "openid email profile",
  "token_type": "Bearer",
  "id_token": "eyJhbGciOiJ...TpsIQ"
}
```

Parameter	Description
access_token	A token that can be sent to the Google API for authentication.
expires_in	The remaining lifetime of the access token in seconds.
refresh_token	A token used to obtain a new access token. Only provided for offline access.
scope	The scopes of access granted by the user (e.g., openid, email, profile).
token_type	The type of token returned. Typically, this is "Bearer".
id_token	A JSON Web Token (JWT) that contains user profile information, such as the user's email address and name, in a secure and compact format.