CS373: Software Engineering
11am, Group 12
November 5, 2020

ConservOcean: Technical Report

Introduction

Covering over 70% of Earth's surface, the ocean is a critical resource for humanity as a resource for food, transportation, and recreation. However, despite its vastness, human activities like overfishing and pollution put the ocean and its inhabitants at risk. Through ConservOcean we hope to create a web application that amalgamates information about these activities and their impacts on specific aquatic species and ocean regions that are impacted. This information will then be dynamically displayed and served through a RESTful API so that people can be made aware of these connections, learn what behaviors are potentially harmful, and act on this information to move towards more sustainable practices.

RESTful API

Our RESTful API is separated into three main categories. The major categories consist of fish, bodies of water, and human impacts. Within each category, we provide additional query parameters and path variables to further refine the query.

For the fish category, the user can return all the instances of fish we have through the /fish endpoint. In addition, the user can also specify attributes that can be used to filter the fish returned. The attributes we support include id, common name, species, location, endangered status, and coordinates.

Similarly, for the bodies of water category, the user can return all the bodies of water available in the database with the /water endpoint. The attributes we support for this category include id, type, name, and latitude and longitude.

The last category, human impact, allows the user to return every type of human impact in the database with the /impact endpoint. The attributes we support for this category include id, category, and latitude and longitude.
For more information on any specific endpoint, please visit the API documentation.

Database

To create our database, we first created our tables using SQLAlchemy in Python. All three of our models connect to each other, with the Water model being the root of any given branch. Water connects to both Fish and Human Impact, with

back-references to Water again to allow for two-way access. Additionally, we implemented a relationship table to have a many-to-many relationship between models that are meant to link to each other.

For scraping other API's to populate our database, we used Python requests to acquire the JSON data of the API call we made, and parsed it to insert relevant data into our database. For endpoints, we used Flask to create web-based endpoints for the frontend to access the data and display it on the website.

Filtering

Backend

For filtering, we used the request parser to obtain query parameters. We then go through all instances and get the ones whose attribute values match those of the query parameters. The instances are returned as a dictionary.

The user is able to filter the instances by using query parameters to specify what they want to be returned. They are able to filter the fish by common name, species, genus, family, endangered status, population trend, and size. Bodies of water can be filtered by type, name, latitude, longitude, size, and water temperature. Human impacts can be filtered by subcategory, latitude, longitude, oil amount (for oil spills), and count density 1 (plastic pollution). For size, latitude, longitude, water temperature, oil amount, and count density, a minimum and maximum range must be given. This is because the values for these attributes tend to vary a lot between instances, so querying for a single value rather than a range wouldn't be practical.

Frontend

In the frontend, we utilized a library called React-Select in order to implement the drop-down menus seen on our model pages. These drop-downs show the user what categories they are able to filter by. Once the "filter" button is clicked, the frontend makes a call to our Conservocean API requesting instances that fit the chosen category. These instances are then displayed on the webpage.

Sorting

Backend

For sorting our database, we mainly used the SQLAlchemy order by method. Majority of our attributes for each model had a predefined ascending and descending order. For instance, "name" was sorted alphabetically and size of the fish was sorted based on numeric value. The only thing that we had to define an ordering ourselves was the endangered status of the fish. We sorted it by how endangered the status is. For instance, we go from "Least Concerned" all the way to "Extinct" for descending.

To specify which category to sort by we used a query parameter called sort. The string passed in with the sort parameter was the attribute to sort by. In addition, we also provided a query parameter called ascending, which was used to determine if the sort was going to be ascending or descending.

For Fish you can sort by common_name, genus, species, status, and size. For bodies of water you can sort by name, latitude, longitude, water_temp, and size. For human impact you can sort by longitude, latitude, subcategory, name, and count_density_1.

Frontend
To implement sorting on the frontend side, we decided to use the same library we used for filtering, React-Select. Again, this was used to create a drop-down menu for users to choose what category to sort and whether they want that sort's result to be ascending or descending. Once the sort button is clicked, the frontend makes an API call for that sort category and displays the results in the grid below.

Searching
Frontend
For searching, we utilized a tool called Algolia, which was recommended to us by another group via a Piazza post. We had to push our backend data to the Algolia server and then use Algolia's built-in Javascript and React commands to query that data in the frontend. The result is a Google-like search interface which returns results very quickly.

For search capability on the model pages, we are still using Algolia again. However, instead of using their frontend searching tool, InstantSearch like we did on the home page, we used a backend search. We decided to do this because it gave us the freedom to style the results on our own. Therefore, ensuring that the search results can look the same as the normal grid on the model pages.

Models
Aquatic Wildlife
This model stores data regarding the creatures that live in the ocean. We provide attributes to each instance of fish such as their scientific name, location, preferred temperature, etc. Instances in this model may have links to instances in the "Bodies of Water" model when referring to the fish's location and may be linked to the "Human Impact" model when referring to common threats the creature faces.

Bodies of Water

This model stores data regarding the bodies of water around the globe, such as oceans, rivers, and streams. Attributes include name, water temperature, salinity, pollution, etc. Instances of this model have links to fish that live in the body of water and to instances of "Human Impacts" that are common in that body of water.

Human Impacts
This model stores data about the human activities that harm wildlife in the oceans. The attributes of this model include category, amount, and affected fish/bodies of water. Instances of this model have links to the impacted aquatic species and bodies of water.

Pagination
The site uses react-paginate to implement pagination. This library provides the user interface for the pagination and stores which page the user of the site is currently on. The functionality of the pagination is implemented by using API calls to our database that obtain all instance pages within a certain numerical range. The range, and consequently the displayed pages, are altered whenever the user changes pages.

Testing
Selenium
Selenium was used to test the UI of the site. The tests open several links on the site to ensure the links lead to webpages with the correct URLs, and checks some of the components of the site to ensure they are on the correct pages.

React Testing Library and ts-jest
The react testing library and ts-jest were used to test the components used to run the frontend of the site. The tests check multiple components to ensure they exist. /*Note that the tests do not currently exist.*/

Python unittest
By using the Python unittest library, we were able to guarantee correctness of all of our API endpoints. We extensively tested return values for most of our query parameters in order to be convinced that our code is proper and will not break on any corner cases.

Postman Unit Tests
Within our Postman API documentation, we created a suite of tests to guarantee that our API calls to the server work properly. We tested return codes, proper formation of entries, and some example values to ensure correctness. Additionally, we added integration with our GitLab repo, so the Postman tests will be executed on every commit of our project.

User Stories - Phase I

The first user story was provided by Marika Murphy and it states, “As a user, I want to see pictures of the aquatic creatures”. This user story wants us to provide images on each fish’s instance page. This request was in line with our planning so it was implemented with the initial setup of the website’s look. The time estimate was around 30 minutes. Putting the images on the webpage was trivial, however, the styling for the images took a little longer. The total implementation time for this user story was about 45 minutes.

The rest of the user stories were provided by Zongying Mo. The first one states “As a user, I want to see pictures of human activities that are impacting aquatic animals”. While images of the activities themselves were not found, we provided images of maps indicating where the human activities took place. If specific images of human activities cannot be found, we will provide stock images for each category of human activity.

The second one states “As a user, I want to be able to find what species are in my state”. This one was out of scope for phase I, because implementing this would require access to the backend API, which has not been implemented yet. After implementing the API the user will be able to see which bodies of water a fish can be found in. Rather than restricting the filtering on the state level, the user can search up the name of the body of water within that state.

The next story states “As a user, I want to see what water a kind of fish lives in”. To accommodate this client preference, a description of the habitat of each species was provided on the instance pages of the fish on the site. In addition, the location where the species is commonly located was also provided.

The last story states “As a user, I want to see if a kind of fish is being fished by humans”. Although we do not have a collective section of all fish being fished by humans, we have an attribute called fishing rate. Fishing rate is a status generated by NOAA (National Oceanic and Atmospheric Administration) that states whether a fish species is overfished. This information can be used for each species to determine whether it is at risk for overfishing.

In addition to the user stories that were provided to us by our customer, we also created several user stories for our developer, homefarmer.me. Our goal with these requests was to document features and data that we would like to see made available from their site as either general users or consumers of their RESTful API.

Our first such request was that “As a client, I would like to see recommended plants to grow in my area.” This relationship between plants and city locations is one of the primary use cases outlined by HomeFarmer’s proposal, so we would expect that it would be achievable in the first phase of this project. The benefit of this feature is that since farming is heavily located and climate-dependent, it is important to know where specific crops can be grown successfully.

In addition to accessing information on plants by region, we wanted access to “information that could help me grow plants. For example, since you have temperature data for cities you could provide information about what temperature is best for the plant.” Knowing about the specific climate details such as precipitation and temperature is critical for most farming. Access to climate conditions by the city would be helpful in knowing what plants a user will be able to grow.

Our third request was “to be able to filter recipes based on the type of food. For example, maybe I want a list of Chinese recipes”. This would allow the user to more easily filter their searches for cuisines they enjoy, which in turn would make it easier to plan a garden with plants they would like. For example, if they enjoy Chinese food, finding out which plants are most common in Chinese cuisine would be easier than searching for various recipes they know they like and searching for plants those recipes have in common.

Our fourth request was to be able to “see nutritional information for recipes so that I can maintain a balanced diet”. This information could be helpful to a consumer who wishes to eat healthier meals, as they can filter out meals that are high in a nutrient they wish to avoid. They may also notice that certain plants are higher in certain vitamins and/or minerals and may choose to add more of those plants to their diets.

Finally, “as an API consumer, I want to be able to request information about recipes that include a certain plant”. For users that have already started growing a certain plant and now have no idea what to do with it, this service would be invaluable. It could also be used to find and consequently eliminate recipes that contain a plant the user is allergic to.

User Stories - Phase II

In this phase, one user wished to know which aquatic animals were impacted by which human impacts, and another wished to know which bodies of water were impacted by a human impact. Both of these issues were solved by adding links from the instance pages of individual human impacts to the pages of the animals and water bodies impacted, which took around 20 minutes each to implement.

One user reported that there were too many words on the page for shortfin squid; this was addressed by changing our source API to one which was less verbose, which took around 1.5 hours to find.

One user wished to find definitions for some terms used to describe aquatic animals, like the “IUCN Population Status”; this information, which was only available on the API Documentation, will be added to a reference page on the website in the near future.

Finally, one user wished for the ability to search for a species by name; this functionality will be added in phase III.

User Stories - Phase III

The first user story asked, “As a fisherman, I would like to be able to filter the fish by size so I can see what large fish I should release back into the water, and which ones I can eat.” This was addressed when implementing filtering with the query parameters size_min and size_max, and took around 15 minutes.

The second user story stated, “As an amateur underwater photographer who lives next to the ocean, I'm interested in being able to identify rare fish that are found in saltwater so I can photograph them when I go diving.” This was handled by implementing the ability to filter fish by population trend and endangered status, and took around 15 minutes.

The third user story asked, “As a boating and fishing enthusiast, I'm looking to go boating on the largest bodies of water possible. I'd like to be able to sort bodies of water by size.” To address this, we added the category to our sort function, enabling this functionality by using the value “size” when providing the sort query parameter. This took us around 10 minutes to implement.

The fourth user story reads, “As a person looking to buy a house, I'd like to live in a place with less pollution for health concerns. I'd like to be able to be able to look at what pollution has occurred filtering by both latitude and longitude.” We solved this by allowing the user to filter human impact entries by location, given a provided minimum and maximum range for both latitude and longitude. This took us approximately 15 minutes to implement.

The fifth user story states, “As a new fishing enthusiast, I'd like to be able to see what kinds of fish I might find in my nearby body of water so that I can research them beforehand.” Our current website has support for this request. To find which fish in a nearby body of water, all you need is to search up the name of the body of water and on the instance page we provide links to fish that can be

found in that body of water. This took us about 2 hours to implement the searching for instances to enable this functionality.

Tools

GitLab

GitLab was used to store the code files that make up our website. It also stores all the important files for this project, including the API documentation and this Technical Report. Our team also used GitLab's Issue Tracking feature for communicating to-do items.

Docker

The front-end development team used Docker to ensure that each member of the team had the same coding environment and toolchain. The image that we created was based on the official Node image, and primarily included the various npm packages that were incorporated into the project, such as React Router and TypeScript. This reduced the difficulty of setting up the environment to improve our collaborative abilities.

React and TypeScript

React and TypeScript were used in addition to JavaScript to code the frontend. We believe that by starting with a framework like React, as opposed to just HTML and CSS, the impending transition from a static to a dynamic site will be made simpler. The additional libraries from React will make it easier to write code without redundancy.

The choice to use TypeScript with our React application ensures proper typing is upheld, which should help to eliminate a whole class of bugs from our codebase. Since TypeScript is a superset of vanilla JavaScript, we will still be able to utilize third-party libraries such as React Router and Mocha without significant additional difficulty.

React Router

React Router was used to implement the various "pages" of our application while implementing the application as a single-page React application with unique URL paths per page. React Router makes site-navigation easier to implement by allowing us to create these navigation elements declaratively. By relying on an external, well-tested library, we are able to add this feature to our site with greater amounts of code reuse and overall reliability.

Mapbox GL JS

One of our main sources for media is maps, so we needed an API that would provide that data. Mapbox seemed to be a good choice for this due to the large amount of free API calls that were provided and the variety of styling options

that were available. Working with Mapbox, we found that the document was high-quality and the implementation was made much easier through the Mapbox GL JS library that worked well with React and TypeScript.

Bootstrap CSS
Our approach to the first phase of this project has focused on data and markup semantics first, and styling second. Using a CSS framework such as Bootstrap allows us to focus on first supplying the content of the site without needing to worry as much about the style since many common components are made available. Additionally, Bootstrap lets us work collaboratively since there is a common set of guidelines and styles that will work well together by default. Eventually, we hope to override Bootstrap's generic appearance with additional, more individualized styling that will improve the look of our site and better convey our overall message.

Node.js and npm
The npm package manager was used to download packages used for the development of our front-end pages. The ability to specify these packages as dependencies, in addition to our Docker image, allows us to better work together by making our programming environments compatible, with fewer bugs due to such inconsistencies. Additionally, using Node.js with React allowed us to make use of the built-in development server so that we could test the design locally before integrating our code with the main repository.

AWS Amplify
We hosted our site through AWS Amplify, which made it quick and straightforward to get our code running on the cloud without the need to set up a back-end for the site. It also simplifies the process of configuring our domain and acquiring an SSL certificate to ensure that the site is available over https. One added benefit of AWS Amplify is that it allows for continuous deployment so that changes to the production site could be made public simply by pushing to the master branch of our GitLab-hosted repository. In Phase 2, we moved away from using Amplify and instead are using EC2. Although it does not have the ease-of-use features of Amplify, we believe that its flexibility will be more beneficial.

EC2
In Phase 2, we switched from Amplify to EC2 for the deployment of our site. Using EC2 makes it simpler to run a Docker container that is portable between our local development and production environments, which allows us to run our Flask backend and React frontend. Our choice of EC2 over products like Elastic

Beanstalk or comparable GCP products was based on its wide range of use cases and support and on our familiarity with AWS systems.

Postman
Postman was used to create the API documentation for the site. We used the collections approach, list of requests, on Postman to create the documentation. Postman will be used to create tests for the API after the implementation in Phase 2.

Prettier
Prettier, much like Python's *black* formatter, allows us to specify and then apply a consistent code formatting across our site. By choosing a consistent standard and automating this style, we can ensure that our codebase has greater readability going forward.

Make
In order to automate our build process, we are using a project-wide makefile that will include rules for building, testing, formatting, and other common tasks. Sharing these commands makes it easier to incorporate additional tooling into our site without the need for remembering individual flags or filenames. Going forward these make commands can also be used in our automated build pipelines for continuous integration and deployment.

Python SQLAlchemy
To create our database, SQLAlchemy was used to create models and links for our API. Additionally, for scraping, the Python *requests* library was used heavily to take data in from other API's, and inserted into our database with SQLAlchemy.

Python Flask
To create endpoints for our database, Flask was used in order to have proper web-based links for the frontend to use to acquire data from our API. By doing this, we have an API that is easily used by frontend to populate our website with information, as well as a robust API that can be used by external users and services, instead of having a crude implementation that is specific to only our deployment.

React Paginate
This library was used to implement pagination. It provides a user interface and an object-oriented interface for the pagination buttons provided on the site. This made it easier to implement pagination, as the developers only needed to implement a page-change function to pass into the react-paginate object and provide the base functionality.

Algolia
Algolia was used to implement searching across all instances (on the home page). It provides a fast searching algorithm based on data we push to the service from our API. It also provides styling and user interface libraries, which we used--in conjunction with our own site's styling--to create the interface of the searching experience.

React Select
This library was used to implement both filtering and sorting. The library provides a user-interface for drop-down selections which we use to show the options a user can sort and filter by.

Hosting
NameCheap was used initially to register the domain name, conservocean.me, for our website at no cost. We were then able to add CNAME and ALIAS records to point any DNS lookups to our site or its www subdomain towards the application content that we are hosting with AWS EC2, which hosts our site running on the Docker image. We were also able to use NameCheap to include redirects for unused subdomains. Hosting our application on EC2 gives us plenty of flexibility, even under the restrictions of the free-tier t2.micro instance. Additionally, we chose to use Amazon Linux AMI as our instance OS, because we believed that it would be the most compatible with the other software that we are utilizing.