CS373: Software Engineering
11 am, Group 12
September 30, 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.

User Stories

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.

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.

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.

Testing
As of now, we have not made use of automated testing methods, since the project has largely revolved around the construction of the frontend side of a static site. Instead, we have relied on informal user testing to evaluate the quality and content of the information that is being displayed on various platforms. Going forward, we plan to create multiple suites of tests using Mocha for our JavaScript frontend and Python's unittest for the application's backend. Additionally, we will use Selenium for end-to-end testing of the site's frontend.

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.

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.

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.

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 Amplify, which automates the process of deploying these DNS records to servers globally with minimal configuration required. Our choice of AWS Amplify over a comparable product through Google Cloud Platform was made largely on the basis of previous experience some of our team members have with AWS.