Critterycovery.me - Technical Report

CS373 - IDB (Phase 3)

William Crawford, Sahithi Golkonda, Shaharyar Lakhani, Daniel Qu, Brian Wang

Table of Contents

Table of Contents	**0**
Team	**1**
Motivation	**2**
User Stories	**3**
RESTful API	**9**
Models	**10**
Tools	**11**
Hosting	**12**
Database	**13**
Backend	**13**
Pagination	**14**
Testing	**14**
Highlighting	**15**
Searching	**15**
Sorting	**15**
Filtering	**16**

Team

Team: Group 16, 11 AM
Website: critterycovery.me
GitLab: https://gitlab.com/cs373-group16/critterycovery
Postman: https://documenter.getpostman.com/view/14742162/TzJrCzVs

Name	EID	GitLab ID
William Crawford	wjc844	WilliamCrawford
Sahithi Golkonda	sg47288	sahithi-golkonda
Shaharyar Lakhani	sl46398	shaharyarlakhani
Daniel Qu	dq764	d.qu
Brian Wang	bw25755	bwang1008

Motivation

Many animals are on the verge of becoming extinct on Earth. In the past decade alone, almost 500 species have gone extinct, and it is unlikely to slow down anytime soon. There are many causes for this, such as deforestation, pollution, and global warming, to name a few.

Unfortunately, this is an issue that rarely receives attention, and most people are apathetic to the condition of severely endangered animals. We want this to change. The problem of endangered species is not something that can be resolved overnight; instead, we have to bring awareness of the seriousness of this problem to public attention so that collective action can be made to save these endangered species. It is our responsibility as living beings on Earth to take care of our home and to make sure every animal has a sustainable environment to live in.

Our goal is to create a website that will raise awareness for endangered animals and help people learn about the habitats and environments that they live in. Educating users about animals that are almost extinct is the first step towards resolving this global problem.

User Stories

Phase I - User Stories Received as Developer

1. This user story wanted to see the most endangered species (those with the smallest populations left). For Phase I, we have put some instance pages of a few endangered species, but we have no method of sorting them by attributes as of now. Currently, we have displayed characteristics of the three models that can be sorted, but the sorting is not implemented yet. We commented on this user story saying we would sort during a later phase, and they changed their request to just viewing some endangered species.

2. This user story wanted to see endangered species in their region. We do have our website have links between instance pages (such as between a species and the region where it's located), but as of now, we do not have filterable requests, nor do we have any endangered species-specific to Austin on our website yet. However, this is a feature that we thought would be great in the final product.

3. The user story wanted to see the diets of endangered species. As of now, some of the APIs we are scraping for data do not contain information about diets, but we will do our best to address this issue. In the worst case, we will change the attributes that are listed on our websites under species.

4. This user story wanted to see images of the endangered species. We are incorporating images of these species into our splash page, the model page for species, as well as our instance pages.

5. This user story wanted to see the wealth of countries as an attribute under our model of countries. We represent this with the GDP of each country.

Phase II - User Stories Received as Developer

1. This user story wanted to see more instance pages for our Species model. In Phase I, we only had three species on our model page. Now, we have almost 500 species. They are organized in a grid manner, with many pages to help with navigation. This ultimately took several days, as we had to set up the database, query from the database, set up the frontend to obtain this information, redesign the model page to display all of our species, and redesign the instance pages.

2. This user story wanted to see more attributes for our Habitats model. This was accomplished by putting more data that we scraped onto our frontend pages. This did not take very long.

3. This user story wanted to see more instance pages and more attributes for our Countries model. In Phase I, we also only had three countries in the Countries tab. Now, we have more than 200 countries displayed, spread out across many pages. We also show more attributes for each country. This also took several days, but we accomplished this relatively quickly after finishing the other two models.

4. This user story wants the units of each information in our table to be put in the header only, rather than in each cell of the table. We fixed this pretty quickly.

5. This user story wants to see a dynamic map for each country. We did this by inserting an iframe that access Google Maps. This took about a day.

Phase III - User Stories Received as Developer

1. This user story wanted us to refine our About page. We did this by centering all of the titles for that page as well as adding icons for all of our tools used. We also changed the cards to be fully scalable with the window size. This took around a day to accomplish.

2. This user story wanted to see more accurate pictures for our Habitats model. This was accomplished by putting more information specific to that habitat into the Google Search API. This took a couple of hours.

3. This user story wanted us to add units to the attributes on our model page tables. This was very straightforward and did not take long.

4. This user story wanted to see nicely formatted instance pages for each species. Specifically, they wanted us to add spacers for information that was not present. This was also fairly straightforward and did not take too long.

5. This user story wanted us to add clear and formatted titles to each of the model pages. This was very simple and did not take long.

Phase I - User Stories Sent as Customers

We have also added our own user stories for the team Around Austin at (https://gitlab.com/jyotiluu/cs373-aroundatx/).

1. Our first user story was that we wanted to access the website through an URL. We wanted the developers to connect the domain obtained from Namecheap with their AWS or GCP server so that their website would be up for public users like us to see. The developer group marked it resolved on March 4.

2. Our second user story was that we wanted to see a homepage on their website. We wanted to be able to access their three model pages from their home page. The developer group marked it resolved on March 5.

3. Our third user story was that we wanted to see a model page for incidences (one of their three models). We wanted to see that it would lead to at least three instance pages and that the model page for incidences would be accessible from the home page. The developer group marked it resolved on March 5.

4. Our fourth user story was that we wanted to see a model page for apartments (another one of their three models). There should be links to some instance pages, and the model page should be accessible from the home page of the website.

5. Our fifth user story was that we wanted to see a model page for restaurants (the last of their three models). There should be links that lead to instance pages, and the home page should have a link to this model page as well.

6. Our sixth user story was that our user stories were not being marked with the Customer label because we did not have the right access permissions (of Reporter).

Phase II - User Stories Sent as Customers

We have added our user stories for the team Around Austin at (https://gitlab.com/jyotiluu/cs373-aroundatx/).

1. Our next user story was that we wanted their API to be ready so that we as customers could pull data from their website. We also wanted to see their Postman documentation for their API.

2. We wanted to see a little explanation in each model page. This way, instead of having the model pages be just for displaying data, users can understand what they are viewing on these pages. This was closed on March 28.

3. We wanted to see pagination on each of the model pages. We did not want to see all of the data being crowded on a single page all at once. This was closed on March 28.

4. We wanted the website to have an icon next to the tab, as well as have a title for the tab. This makes the website feel more legitimate, rather than the website having default values in the tab. This was closed on March 23.

5. We wanted each instance page to have more media, like pictures, maps, news, etc. This way, users of the website will be drawn more to the information presented. We also wanted to see actual stars for the hotels' ratings. This was closed on March 29.

Phase III - User Stories Sent as Customers

We have added our user stories for the team Around Austin at (https://gitlab.com/jyotiluu/cs373-aroundatx/).

1. Our first user story for this phase was the issue of scaling on their website. Much of the text became unreadable when the window size became too small or if someone was viewing it on a mobile device. This was closed on April 15.

2. Our next story requested the implementation of searching through hotels, restaurants and incidents. We also requested them to implement a search page where the results are shown. This was closed on April 16.

3. We wanted them to address and fix a bug we had found in their website involving the back button. When someone navigates to the model page and moves to another page of the results via pagination and clicks on an instance page, the back button will navigate them back to the beginning of the table or grid. We wanted this issue to be solved. This was closed on April 16.

4. As users, we also wanted to be able to sort the various models by their attributes. This would greatly help find the extreme values such as the cheapest or most expensive restaurants. This was closed on April 13.

5. For the last user story, we wanted their Hotels model page to be properly centered. We also wanted them to fix a bug where their pagination would sometimes display “1-0 of 0” rather than “0-0 of 0.” This was closed on April 14.

Restful API

The Postman is located at https://documenter.getpostman.com/view/14742162/TzJrCzVs. We defined some GET requests that we would like to see implemented because they would be useful. In particular, we defined the following:

Countries:

- `https://critterycovery.me/api/countries:` Retrieves all countries
- `https://critterycovery.me/api/countries/name=<name>`: Given the name of a country, this retrieves all attributes for this country
- `https://critterycovery.me/api/countries/alpha2_code=<alpha2_code>`: Given the ISO 2 code of a country, this retrieves all attributes for this country
- `https://critterycovery.me/api/countries/alpha3_code=<alpha3_code>`: Given the ISO 3 code of a country, this retrieves all attributes for this country
- `https://critterycovery.me/api/countries/habitats/name=<name>`: Given a country name, this retrieves all habitats within this country
- `https://critterycovery.me/api/countries/species/name=<name>`: Given a country name, this retrieves all endangered species within this country

Habitats:

- `https://critterycovery.me/api/habitats`: Retrieves all habitats
- `https://critterycovery.me/api/habitats/name=<name>`: Given the name of a specific habitat, this retrieves all attributes for this habitat

Species:

- `https://critterycovery.me/api/species`: Retrieves all species
- `https://critterycovery.me/api/species/name=<name>`: Given a scientific name of a particular species, this would retrieve all attributes of this species
- `https://critterycovery.me/api/species/countries/name=<name>`: Given a scientific name of a particular species, this would retrieve all countries that this species is documented to live in

Models - Phase 2

- Species

  This model now dynamically fetches data from our RESTful API and includes all species instance pages that can be reached from the species main page. The species main page now includes a grid with pagination. Some examples of species are the Asiatic Cheetah, Cuetzalan Salamander, and the Guatemala Stream Frog. The instance pages for the species include an image for each species as well as the taxonomy and a description of the geographic range and population. It also includes links to the countries it is native to, and the habitats it lives in.

- Habitats

  For our habitats, we are also now using data fetched from our RESTful API and displaying the information in a table. The habitats main page also includes pagination. From the main page, we can access all instances of the habitats which include information about the land and water areas and what type of designated site it is. It also includes links to the species that live in that habitat and the countries that it appears in.

- Countries

  For our countries model in phase 2, we used a RESTful API and dynamically fetched data to populate our table. We currently have all countries and recognized territories in our paginated table, however, we plan on removing countries that do not have any recognized habitats or endangered species living in them. Within each instance page of the flag model, information about the population, capital, and regions is included. We also include an image of the flag and an interactive Google Maps extension that shows where the country is located. Additional links also show which species and habitats are present within this country.

Tools

- GitLab: Our repository is located at (https://gitlab.com/cs373-group16/critterycovery/-/tree/master). In a later phase, we will use GitLab's continuous integration to run automated tests. We divided our project into the backend and frontend folders.
- React Bootstrap: We are using React Bootstrap to stylize our website. We used Cards for our About page, Tables for our model pages, and React-player for our splash page.
- AWS EC2: We are using an AWS EC2 instance to host our website.
- AWS Route 53: We used AWS Route 53 to redirect "critterycovery.me" to our actual website.
- Google Cloud Platform: We used this tool to host our database. We created a script to populate the table, which we ran once. The API endpoints access the remote populated database, then format it for our own uses.
- Postman: We used Postman to query from our various APIs, as well as create our API.
- TypeScript: We are using React with Typescript so that we can avoid type errors down the road.
- Flask: We used Python Flask for our backend and API calls.
- Docker: We are using Docker so that the EC2 can pull from our repository and run a container to host our website.
- Selenium: We used Selenium with Python to test the GUI of our website.
- Jest: We used Jest to test our React with Typescript for the frontend.

Hosting

Our site is being hosted as an AWS EC2 instance that will start a Docker image that hosts both our frontend and backend. The HTTPS authentication is provided by AWS Route 53. Our domain name of critterycovery.me was registered with Namecheap.

When first starting, we obtain a file with the extension “pem” to be able to remotely access the EC2 instance through ssh. The purpose of using EC2 is that even when we log out of the EC2 instance, it will continue to host the website.

Initially, the EC2 instance Amazon gave us was blank. For it to run our website, we downloaded Docker on it, so that it could run an image and host our website. The first step after accessing the EC2 instance was to install docker. Then we cloned our repository hosted on GitLab to the EC2 instance, and ran the Dockerfile in our repository with “docker build”. This file has all the commands needed for docker to set up the website. First, the file pulls from the master branch of the repository to get the latest version of our code. Next, the file runs “yarn install”, installing all the dependencies for the project and “yarn build”, which creates a static folder for the website to use for the frontend.

Finally, the EC2 instance starts our website with “docker start” followed by the name of our built image. This means our website (critterycovery.me) is accessible to the outside world!

Database

We used GCP for our database. We created a script to pull from various API's and create tables for our models in our database. The API responses returned JSON objects, and we parsed through each output and scraped what we wanted. Each of our API's provided endpoints for getting a lot of information about each instance in each model, and as a result, our tables were each able to have a bunch of columns for different attributes. Currently, we have 4 tables. One for each model, and one linking table for the connections from species to habitat to make our api easier to implement. We connected to the Cloud instance with Python's SQLAlchemy library. To do so, we had to use the following line:
`create_engine(f"postgresql://{db_user}:{db_password}@{db_name}")`, where the appropriate variables are obtained from when the Google Cloud Platform was set up. Once the tables in the database were created, we were able to use the flask-restless library to create our own API, which we are calling from the frontend.

Backend

We implemented our API with Python Flask and Marshmallow in api.py. The goal is that the frontend should request data from our backend to populate the pages, and the backend should request data from the populated database, process the data, and then return it as a JSON object to whoever called that API (like the frontend). We created Marshmallow Schema classes mirroring our own models. Then in one of our API methods, we used our model to query the database and used a Schema instance to dump (serialize) the information. We add the decorator @app.route(URL) so that the information is obtainable at the URL. With the API endpoint set up, the frontend uses axios.get(URL) to obtain the information. In the way we set up Google Cloud Platform, it was necessary to add all of our IP addresses so that we had access to the database.

Pagination

We originally used the npm library react-paginate, but now we are using Paginate from react-bootstrap. In our Pagination component, we computed how many pages there would be based on the total number of instance pages and the number of instance pages per page. We used a for loop to add the clickable buttons (Pagination.Item) onto the pagination bar, depending on if the page number was positive and less than the total number of pages. The cards that are to be loaded for a specific page depend on the page number and the number of instances per page. The back and next buttons are implemented based on the current page and the number of instances per page too. We implemented pagination for all three of our model pages.

Testing

We used Python's unittest library for testing the backend and our Python code. In our backend/python_tests.py, we used Python's requests library to query our website and make sure the information on the website is accurate. For instance, we make sure that the number of instances of each model is correct.

We used Selenium in Python to test our GUI on the website. This is in tests/selenium. An actual webdriver needs to be installed, so we included one on our GitLab repository. After setting up a few options (like headless mode to prevent a browser from popping up), the Selenium webdriver navigates to our website, clicks on a few links through our website, and makes sure that critical information is there and is where it needs to be.

We also used Jest to test our frontend and React. We also used Enzyme to help test our React components. We used Jest to create snapshots of our React components. Then the test compares the current components with the saved snapshots.

We used Postman to test the data in our database. We added several tests on the Postman website, before exporting it into a JSON file that we added to our repository. We used Newman in our pipeline to test this file.

Highlighting

We decided to replace our tables from react-bootstrap in our Country and Habitat model pages with Ant Design's tables because we could not get highlighting working. Ant Design's table item renderer will be called whenever the user wants to sort or filter the data, unlike react-bootstrap's tables. We also used a Highlighter component from react-highlight-words to display the appropriate word found. In the tables, this highlighter is called each time the table data gets searched. For our cards version, the highlighter is inside the card, and whenever the searchVal prop gets updated, the highlight gets applied.

Sorting

We wanted to minimize the number of calls to our backend, so we have sorting done in the frontend. For the cards in our grid (the Species model pages), we manually call sort on our list of cards before rendering them on the model page whenever the sort selection input is changed using Javascript sorts on arrays. (see frontend/src/components/CardDecks/SpeciesCard.tsx). For the Ant tables, we just had to add a comparator to the appropriate columns to allow sorting, since these tables have this functionality built-in.

Searching

We also do searching in the frontend. For our model pages, we added a Search sub-component of Input from Ant Design for the users to type in a query. A React hook then takes the query, and filters the data to match it by seeing if the term is in any of each item's attributes.

For searching on the Nav bar / all models, we created a new page for search results. We run the same hook on each instance (each a different component), and display them on the Search page. That search page links all of the results to their respective models.

Filtering

For our table model pages, we added dropdowns so that users could have room to filter the data other than the table header. There are two types of filters for tables: text and number. The number filter is only different from the text by adding a select dropdown for number comparisons and filtering whether that comparison is true. These options included less than, greater than, equal, and not equal to. Whenever the user clicks the submit button, a submit function is called which applies the filter on the correct column by using a current column and current input var.

For the grid model page, filtering is available to the users through an input component for each filter. Whenever a user inputs something into one of the filters, the corresponding filter value gets updated, and the filter() function is called. That filter function applies a Javascript array filter to the sorted data and outputs the final data.