Technical Report for Group 22

Endangered Animals

Table of Contents

Motivation

The issue of endangered animals and species extinction has always been a major concern of animal conservationists and environmentalists alike. Unfortunately, throughout the years, there has been less and less care and concern for the protection and preservation of endangered animals. We chose this topic in order to promote and spread awareness of this issue. This task will be accomplished by making information about endangered animals and their threats more widely available. The information will be retrieved by

making several RESTful calls to various RESTful APIs. Afterwards, the data will be aggregated, compiled, and displayed on our website, which can be accessed at endangered-animals.me.

Use Cases

The user can interact with our web app in multiple ways.

The primary function of this website is to provide information on endangered animals. For example, let's say a user wanted to find out information about the African Elephant, an animal that is currently at risk of extinction. The user could look up the animal on the Animals page, which would return pertinent information about the animal, such as its current vulnerability status, habitats that the animal lives in, threats currently affecting the animal, and conservation measures taken to ensure its survival.

Users can also utilize our website to find information about specific habitats.
For example, let's say a user wanted to find out information about endangered animals living within temperate forests. The user could look up the habitat on the Habitat page, which would return all endangered animals living within the habitat, specific threats that affect that habitat, as well as the overall suitability of the habitat.

Thirdly, users can utilize our website to find information about specific threats.
For example, let's say a user wanted to find out information about how agro-industrial farming endangers animals. The user could look up the threat on the Threats page, which would return all endangered animals affected by the threat, a list of habitats that are affected by the threat, the severity of the threat, and whether the threat is ongoing or not.

Lastly, users can utilize our website to find information about specific countries.
For example, let's say a user wanted to find out information about endangered animals in the country of Botswana. The user could look up the country on the Country page, which would return all endangered animals that currently live in the country, a list of habitats that are within the country, and other information such as the name and geographic location of the country.

RESTful API

Our API centers on four data models: Animals, Habitats, Threats, and Countries.
Each model will have its own list of RESTful operations (GET).

Animals APIs

There are two APIs to get the fields for the animal model: /single_animal_data, /all_animal_data

GET /single_animal_data/?animal_name=NAME

Given the common name of an animal, this API returns a JSON object filled with the following fields: the animal's scientific name, the animal's common name, the animal's vulnerability status, a citation link for information on the animal, a web link to the animal's detailed report page on Red List's official website, a YouTube link to a video of the animal in action, the animal's conservation measure (action taken to conserve the species), habitats that the animal lives in/ its associated habitats, threats that the animal faces/ its associated threats, and countries that the animal lives in/ associated countries.

For example, given the animal "Semi-collared Hawk", calling this API would return a JSON object with an output of {"common_name": "Semi-collared Hawk", "scientific_name": "Accipiter collaris", "vulnerability_status":"vulnerable", ...}

GET /all_animal_data

This API returns a list of JSON objects containing the data for all animals. This API takes no parameters. Calling this endpoint returns the equivalent of calling the endpoint "/single_animal_data" for each animal in our animal database.

Threats APIs

There are two APIs to get the fields for the threats model: /single_threat_data, /all_threat_data.

GET /single_threat_data/?threat_name=NAME

Given the name of an threat, this API makes various calls to Red List IUCN's API, and returns a JSON object filled with the following fields: the name of the threat, the timing of the threat (whether the threat is ongoing or not), the severity score of the threat (a number from 1 - 10), a list of animals affected by this threat, and a list of habitats that this threat is present in.

For example, given the threat "Agro-industry farming", calling this API would return a JSON object with an output of {"name": "Agro-industry farming", "image": "https://vermont4evolution.files.wordpress.com/2011/07/mono1.jpg", "timing": "ongoing","severity": "4.5", ...}.

GET /all_threat_data

This API returns a List of JSON objects containing the data for all threats. This API simply calls the endpoint "/single_threat_data" for each threat in "/single_threat_data".

Habitats APIs

There are two APIs to get the fields for the habitat model: /single_habitat_data, /all_habitat_data

GET /single_habitat_data/?habitat name=NAME

Given the common name of an animal, this API makes various calls to Red List IUCN's API, and returns a JSON object filled with the following fields: the name of the habitat, an image of what the habitat looks like, the habitat's suitability status/the extent to which a habitat can sustain the animals that live within it, a list of associated animals that live in the habitat, and associated threats for a habitat.

For example, given the habitat "Artificial/Aquatic - Ponds (below 8ha)", calling this API would return a JSON object with an output of {"name": "Artificial/Aquatic - Ponds (below 8ha)", "image": "http://thumb1.shutterstock.com/display_pic_with_logo/1771478/323878760/stock-photo-small-artificial-decorative-pond-with-rocks-and-plants-on-the-backyard-in-summer-323878760.jpg", "suitability": "Suitable", "assoc_animals": ....}.

GET /all_habitat_data

This API returns a List of JSON objects containing the data for all habitats. This API simply calls the endpoint "/single_habitat_data" for each habitat in "/all_habitat_names".

Countries APIs

There are two APIs to get the fields for the country model: /single_country_data, /all_country_data

GET /single_country_data/?country name=NAME

Given the common name of an animal, this API makes various calls to Red List IUCN's API, and returns a JSON object filled with the following fields: the country's name, an image of the flag of the country, a pair of global coordinates representing the country's geographical location, a list of animals that live within the country's borders, and a list of habitats that lie within the country's borders.

For example, given the country"Azerbaijan", calling this API would return a JSON object with an output of {"name": "Azerbaijan", "flag": "http://cdn.wonderfulengineering.com/wp-content/uploads/2015/07/azerbaijan-flag-13.png", "coordinate": "lat: 40.143105 lng: 47.576927", ...}.

GET /all_country_data

This API returns a List of JSON objects containing the data for all countries. This API simply calls the endpoint "/single_country_data" for each country in "/all_countries_names".

Models

Our website utilizes 4 models: Animal, Habitat, Threats, and Country. Each model contains at least 2 connections to 2 other models, denoted below by asterisks.

Our models pull from 4 RESTful API sources:

1. IUCN Red List API
2. Google's Youtube Search API
3. Google's Geocoding Coordinates API
4. Bing's Image Search API

Animals

The Animal model represents an endangered species. There is a webpage called "Animals", in which the user is able to view all animals or search for a specific one. The Animal model contains the following fields:

- Common name of the animal
- Scientific name of the animal
- Image of the animal
- Vulnerability status
- Conservation measures taken to preserve the animal
- Citation link for the original report on Red List
- Video link showing the animal in action if available, retrieved from YouTube's API
- *Threats this animal faces (associated threats)
- *Habitats this animal lives in (associated habitats)
- *Countries this animal is present in (associated countries)

This model depends on YouTube's API for the video link, and Bing's API Images Search API for the image link. All other data comes from IUCN Red List's RESTful API.

Threats

The Threats model represents a source of danger for a species. There will be a page called "Threats", in which the user is able to view all threats or search for a specific one. The Threat model contains the following fields:

- Name of threat
- Timing: a measure of whether the threat is an ongoing threat or not
- Severity score (from 1 - 10)
- *Animals affected by the threat (associated animals)
- *Habitats affected by the threat (associated habitats)

Habitats

The Habitat model represents an ecosystem that endangered animals live in. There will be a page called “Habitat”, in which the user is able to view all habitats or search for a specific one. The Habitat model contains the following fields:

- Name
- Image
- Sustainability status: whether it can adequately support organisms
- *Animals that are found in this habitat (associated animals)
- *Countries that this habitat is associated with (associated countries)

This model depends on Bing’s Image Search API. All other data comes IUCN Red List’s RESTful API.

Countries

Lastly, the Country model represents a nation. There will be a page called “Countries”, in which the user is able to view all countries or search for a specific one. The Country model contains the following fields:

- Name
- Flag image
- Coordinates pair for global position
- *Animals that live in this country (associated animals)
- *Habitats that are within this country (associated habitats)

This model depends on Bing’s Image Search API for images and Google Maps’ Geocoding RESTful API for coordinate information. All other data comes IUCN Red List’s RESTful API.

Database (DB)

The backend for this application consists of 2 parts:

1. A MySQL database hosted on a Cloud SQL instance.
2. A RESTful API made from Flask, which uses SQLAlchemy.

MySQL Database

All of our data is stored on a MySQL database, which is hosted on a Cloud SQL instance. This Cloud SQL instance is connected to our Google Cloud Platform (GCP) App Engine project.

Flask-SQLAlchemy RESTful API

We used a combination of Flask and SQLAlchemy to create our RESTful API. By writing the following lines of code, we were able to set up our application and create all of our database tables:

Inside models.py

```
app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = os.environ['SQLALCHEMY_DATABASE_URI']
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
engine = create_engine(os.environ['SQLALCHEMY_DATABASE_URI'], echo=False)
db = SQLAlchemy(app)
```

...where SQLALCHEMY_DATABASE_URI is a field inside our app.yaml file, describing our connection string to our sqlalchemy database. It is in the format: mysql+pymysql://USER:PASSWORD@/DATABASE?unix_socket=/cloudsql/INSTANCE_CONNECTION_NAME

(See https://github.com/GoogleCloudPlatform/python-docs-samples/tree/master/appengine/flexible/cloudsql for more information)

Inside main.py

```
if __name__ == '__main__':
        app.config['SQLALCHEMY_DATABASE_URI'] =
'mysql+pymysql://root:123@127.0.0.1/SWE_DATABASE'
        app.run(debug=True)
```

Pushing data to our database

Since SQLAlchemy is an ORM (Object-Relational Mapper), we can add objects into our database using the following lines:
Given a class habitats, and a habitats object h1:

```
h1 = habitats(“name”, “imageUrl", "suitability")
db.session.add(h1)
db.session.commit()
```

Adding Many - Many relationships between objects

In order to set up the associated tables, we had to set up relationships between pairs of objects. For example, we want to set up the relationships between these 3 related tables: animal, country, and associated_countries_animals. The animal has a 1 - Many relationships with associated_countries_animals, and Country has a 1 - Many relationship with associated_countries_animals as well.
The primary key for the animal table is a foreign key inside associated_countries_animals, and the primary key for the country table is also a foreign key inside associated_countries_animals. The following code sets up the relationships between these tables:

Inside animal table

```
associated_countries_animals_link = db.relationship( 'associated_countries_animals', backref='animal', lazy='dynamic')
```

Inside country table

```
associated_countries_animals_link = db.relationship(
        'associated_countries_animals', backref='country', lazy='dynamic')
```

Inside associated_countries_animals table

```
scientificName = db.Column( db.String(255), db.ForeignKey('animal.scientificName'))
countriesName = db.Column(db.String(255), db.ForeignKey('country.name'))
```

Database Tables

Our database contains 9 tables:

- 4 model tables, and
- 5 “associated” tables that connect each of the 4 tables.

The 4 model tables are:

- animal
- threats
- habitats
- country

The 5 “associated tables” are:

- associated_countries_animals
- associated_habitats_animals
- associated_threats_animals
- associated_habitats_countries
- associated_threats_habitats

Each of the associated tables connects 2 of the model tables together. These tables were created because we needed to find a way to store lists inside a database, without violating the fundamental database property of one value per column. For example, for each animal, we had a list of countries that the animal was present in. However, we couldn’t store that in a column. In order to circumvent this, we simply select the countries from the associated_countries_animals where the name is the animal’s name.

Search

A key component to any website is allowing users to manipulate the data they view on the front-end. This includes features like filtering, paginating and sorting data, as well as searching for data. These processes are typically complex and require much coding on the developers' side. Our project has focused instead on lifting up much of this work for faster development.

Our code repository relies on another React-bootstrap framework repository. We also make use of another React repository which contains the implementation for a component called React-Bootstrap-Table. This table component has features embedded within it to allow specified columns to automatically be sorted by values, ascending or descending. The table component also has an interface for filtering data based on model instance attributes and/or searching through the available data with filters applied.

Tools

GitHub: GitHub is the platform we used to manage our code. It handles source control and allows our team to work on different branches. The most relevant use case for this is adding on experimental features on a separate branch from the live website itself, allowing the development team to perfect the experimental code before deploying it to the entire Internet. GitHub also contains other useful features, such as comparing changes across commits, helping us to meticulously review each other's code to ensure safety.

Trello: Trello's purpose for this specific project is to explicitly define subtasks of our development and clearly assign who will work on what. Trello splits tasks into three categories: need to be done, doing, and done. Because a development team cannot meet as one unit at all times, Trello resolves gaps in communication by keeping all members up-to-date with the latest progress. Trello is essentially a real-time "sticky notes board."

PlantItPoker: PlanItPoker is a tool to help manage some of the actual completion of the Trello tasks. We first create user stories to model what users should expect to see when coming to our website and estimate their interactions with the interfaces. This helps developers understand that the ultimate focus of development is to serve the user, i.e., keeping a user-centered mindset.

Additionally, PlanItPoker allows a team to vote on the estimated amount of time it should take for each user story to be completed. User stories can roughly be correlated with Trello tasks. Teams vote anonymously on the estimated time for task completion, helping to voice each developer's ideas in a safe environment.

At the end of each development phase, PlanItPoker can be updated to reflect the actual total time spent on each task. Long-term, this helps developers understand the discrepancy between our perceptions and reality, and aims to help the team as a whole estimate future projects

accordingly. In this way, PlanItPoker serves to accomplish three main functions for the development team.

Bootstrap: is a CSS, HTML, and JS framework that helps in developing a responsive web application. Bootstrap allows us to rapidly prototype our website without getting bogged down by making sure our website is responsive in all browsers/devices. Bootstrap is currently directed integrated with our React framework to ensure the best compatibility.

React: React is the Javascript library used to build the front-end user interface (UI). React has a lot of advantages, namely integrating Javascript directly into HTML elements. We specifically relied on a React-Bootstrap template that is based on React components. This allowed us to modularize each part of the website (e.g., navigation bar, footer, etc.) to ensure a change made to one component is reflected globally. With React managing the UI, we only needed to focus on translating our back-end data into an effortless and meaningful interface for users.

Flask: Flask is a micro-web-framework written in Python, which is an excellent programming language for rapid high-level application development that offers tons of useful libraries. Flask is easy to get started with as a beginner because there is minimal boilerplate code required for getting a simple app up and running. We have used Flask to build our website server to handle RESTful APIs. With Flask, building web services is surprisingly simple and it works perfectly with our cloud database. Flask-SQLAlchemy, an extension provided by Flask, provides a wrapper for the SQLAlchemy project, which is an Object Relational Mapper (ORM). Flask-SQLAlchemy allows our database to work with objects instead of tables and SQL. We don't have to learn SQL to build our database because we already let Flask-SQLAlchemy do the job for us. For example, we use db.createAll() to create our tables. With just that line of code, it runs against our models and creates the necessary tables if they don't already exist. Flask abstracts much of the database work, allowing for faster app development.

Apiary: Our project's central goal is to combine several different data sets in meaningful ways for a new interface. If other development teams wish to utilize our curated data, we have exposed RESTful API calls from our server for these development teams to utilize. Apiary is essentially the documentation for understanding how to access our API calls and back-end data. We recommend other teams also utilize Apiary while in development as it helps the back-end team communicate to the front-end team the necessary calls the front-end can utilize for the successful completion of the project.

MagicDraw: MagicDraw is a visual UML modeling tool with team collaboration support. Its main purpose is to help consolidate complicated databases in concise and telling diagrams, helping to keep the entire development team clear about the specific implementation of the back-end. This proves to be useful for front-end developers as well, as they can look at different tables on the back-end and discover different ways of combining data sets for new insights. Designed for software analysts and programmers, this dynamic and versatile development tool facilitates analysis and design of Object Oriented (OO) systems and databases.

Postman: When programmers want to create a Web API, they have to test their JSON API on at least two environments: local and staging. So, programmers have to create two request_form's with each API but the only difference is the base_url. It is very inconvenient since programmers have to copy/paste too many times. Postman eliminates those problems, by allowing us to set up multiple environments to switch context. It saves us time when we test our code and serves to write test our unit tests to ensure our back-end APIs are operating as expected. The API request can be created just under 20 seconds each time.

WampServer: WampServer is a utility designed to allow us to create a web application and manage our server and database. WAMP is an acronym formed from the initials of the operating system Windows, Apache, MySQL, and PHP. WampServer allows users to set up a server locally on their Windows machine to create dynamic web applications with Apache, PHP, and MySQL database. However, our team only uses WampServer to create and interact with the database. As the result, we did not have to wait for more than 10 minutes every time if we found a bug and we had to update the database.

Slack: Slack is an effective teamwork tool that allows for synchronous communication on different devices. We've split our team into backend and frontend channels, as well as additional channels to manage meetup times and file sharing. Slack supports on Android and iOS so team members can receive notifications on the go. We can check the message with our browsers or with our phone when we are on our way to class. Slack is extremely convenient since we are working in a group of five and not all of us are using the same operating system.

Hosting

Application Requirements

Pre-requisites:

1. Git
2. Google Cloud SDK
3. Python3
4. Flask

In your local machine, create a folder for the project and clone its source from this Github repo: https://github.com/gvikei/idb

Running the front-end locally

- cd to “front-end” within the project root folder
- For the first time, run “npm install” to install all dependencies
- Once all dependencies are installed, run “npm run docs” to launch the server
- Open “http://localhost:8080” in a browser to see the result
- Any changes made will trigger an auto-recompile. The website need only be refresh in browser

Deploying the front-end

- For deploying the front-end, instead of using Google App Engine, we go with Google Container Engine instead.
- Navigate to the root folder of the project, which contains the “Dockerfile” file.
- From this Dockerfile, we built the Docker image with:
    - docker build -t gcr.io/crafty-cairn-184716/idb .
- Next, we test the whether the Docker image works locally by using:
    - docker run --rm -p 1802:8080 gcr.io/${PROJECT_ID}/idb
- We then push the Docker image to Google Container Registry by using the following command
    - gcloud docker -- push gcr.io/crafty-cairn-184716/idb
- Next comes creating a container cluster called hello-cluster to run the Docker image
    - gcloud container clusters create hello-cluster --num-nodes=1
- To deploy the app and have it listenning on port 8080, we use the following command:
    - kubectl run idb --image=gcr.io/${PROJECT_ID}/idb --port 8080
- After that, we explicitly expose our app to traffic from the Internet, since the app isn’t accessible from the external at the beginning:
    - kubectl expose deployment idb --type=LoadBalancer --port 80 --target-port 8080
- To get the external IP for our app, we run “kubectl get service” and got the following result:

```
NAME         TYPE           CLUSTER-IP     EXTERNAL-IP      PORT(S)        AGE
idb          LoadBalancer   10.11.249.221  35.193.177.152   80:32673/TCP   35m
kubernetes   ClusterIP      10.11.240.1    <none>           443/TCP        42m
```

- Now what remains is just to point our current domain to the external IP.

Updating changes

- Everytime we make changes to the site and want to deploy it, we have to create a new Docker image, then push it to the Container Registry and finally update the image as follow:
    - kubectl set image deployment/idb idb=gcr.io/${PROJECT_ID}/idb

Testing the back-end locally

- In order to create and test our database, we used WampServer and Postman
- WampServer will create a localhost on our machine. To connect it that localhost, we enter: "http://localhost/phpmyadmin/index.php". It will take the user to the list of databases on the machine
- When we can confirm that the database is appropriately created on the localhost. We can begin to test our APIs with the help of Postman
- For the second phase of the project, our APIs only interact with the database using GET so we choose GET in the "Params" field to test our API locally
- We run our Python
- In the "Enter request URL" field, we enter the "http:127.0.0.1:5000/API_route" where API_route is the route created in our python file.
- Hit "Send" button to run the API. The API should return a JSON file.
- We also create IF statement to check for the user input. If the API_route parameter is empty, it should raise an error and require user to input the valid route. In this case, user has to enter valid animal's name, country, habitat, or threat. We also check the case when the database goes offline. It also raises an error.
- If no error is raised, we can check our output data by using "Body" tab and choose "JSON". The JSON file should not be empty and has multiple fields depending on the API_route.

Deploying the back-end

- Navigate to the project root folder. This folder should contain (at least) the following 3 files:
    - apps/models.py - File containing all the classes for the database tables
    - back-end/main.py - File containing REST api
    - back-end/app.yaml - Deployment file containing sqlalchemy database uri
- Make sure you have GCloud SDK tool installed, by typing "gcloud --help"
- Login and configure settings by typing "gcloud init"
    - Configure the correct email
    - Configure the correct project

    - When it prompts to connect to Compute Engine, exit.
- Set your account through the command “gcloud config set account ACCOUNT” where ACCOUNT is your e-mail
- Login to gcloud through “gcloud auth login”
- Set your current project through “gcloud config set project PROJECT”
- Deploy your files by typing in “gcloud app deploy”. This should typically take anywhere from 5 - 15 minutes.
- The base url for all the RESTful API calls will be “https://swe-endangered-animals.appspot.com/”. Refer to our Apiary for endpoints.

Adding a custom domain

Namecheap is the service we used to reserve our domain name. The best thing about Namecheap is it's free for students as long as they have a .edu email account and the domain name has to be ended with .me. The setup is very straightforward. We only searched for the domain we want, and Endangered-animals.me was assigned to our group since no one has taken that name. Moreover, we can keep that domain for a year, which will be plenty of time for our group to finish the project and use it as one of our references to impress recruiters in the future.

- We registered the domain www.endangered-animals.me at Namecheap
- Navigate to https://console.cloud.google.com/appengine/settings/domains and click “Add a custom domain”
- Click the dropdown “Select the domain you want to use option”, and choose “Verify a new domain”
- Enter the newly registered domain into the textbox “mydomain.com” and click “Verify” and the “Webmaster Central” page will be opened up.
- Click the “Add a Property”, type in your Namecheap.me domain and click “Continue”
- Copy the TXT record given in the Webmaster Central
- Open Namecheap Dashboard, click “Domain List” and click “Manage”
- Go to the “Advanced DNS tab” and add a TXT record with the following info:
    - Type: TXT Record
    - Host: @
    - Value: the TXT record given by Namecheap
- Go back to the Webmaster Central and click “Verify”
- Once the custom domain is verified, go to https://console.cloud.google.com/appengine/settings/domains, click “Add a custom domain”, choose the target domain from the dropdown list and click “Continue”.
- Save the default mappings and choose to config resource records option, you’ll be given a set of resource records at this point

Go back to the “Advanced DNS” tab in Namecheap and add all those records given by Google in.