# CS 340 README

## About the Project

This project aims to develop a comprehensive software application for Grazioso Salvare, an innovative international rescue-animal training company. Grazioso Salvare specializes in identifying and training dogs for search-and-rescue operations, crucial for aiding in life-threatening situations.

The application will facilitate the identification and categorization of dogs suitable for search-and-rescue training. It will utilize existing data from the partnered non-profit agency, focusing on specific dog profiles such as age, breed, and potential rescue proficiency. The dashboard will serve as the client-facing component, allowing users at Grazioso Salvare to access and manage the database efficiently.

## Motivation

The motivation behind this project comes from the critical role that search-and-rescue dogs play in emergency response scenarios. By creating a user-friendly dashboard and database interface, the project aims to streamline the identification and categorization of potential search-and-rescue dogs, ultimately contributing to more effective rescue operations and saving lives. The decision to make the project open source reflects Grazioso Salvare's commitment to collaboration and knowledge sharing within the rescue-animal training community, fostering innovation and advancement in the field.

## Getting Started

To get started with this project, follow these steps:

1. Download/fork and clone this GitHub repository to your local machine. This repository includes animal\_shelter.py, ProjectTwoDashboard.ipynb, Grazioso\_Salvare\_Logo.png.
2. Extract the repo if you downloaded the project as a zip file. Open ProjectTwoDashboard.ipynb using Jupiter Notebook or any appropriate IDE.
3. Import the data set you want to use for the project.
4. Create a Mongodb user account so you can connect with the databases if you haven’t done so.
5. Follow the usage instructions in this README to run the application, test or to modify as per your requirements.

## Installation

Ensure you have Python, MongoDB and Mongo DB Shell installed on your system. In order to import the data set into MongoDB, you also need to use the mongoimport tool. Besides, you need to use Jupiter Notebook or equivalent tools to run the code and start the Dashboard application.

## Usage

### *Code Example*

The first step is to import the dataset. In this example project, we will utilize collection animals in the AAC database. First, import a data set into MongoDB using the following command:

*mongoimport --username="${MONGO\_USER}" --password="${MONGO\_PASS}" --port=${MONGO\_PORT} --host=${MONGO\_HOST} --db AAC --collection animals --type csv --authenticationDatabase admin --drop --file fileName.csv –headerline*

Replace fileName with the name of the CSV file you wish to use as the dataset for the project. If the dataset already exists, you may skip this step unless you intend to import a new dataset. Note that the *--drop* command will discard the existing data and replace it with the new file you are importing.

The second step is to create new user (if needed) in database “admin” and a role in database “AAC” with a command line in mongosh:

*db.createUser({user: "your\_username", pwd: "your\_password", roles: [{role: "readWrite", db: "AAC"}]})*

You can use these credentials for the next steps in the project. To view the Dashboard UI, open the "ProjectTwoDashboard.ipynb" file in Jupyter Notebook and click "Run" from the menu bar at the top of the file. This action will generate a link at the bottom of the notebook. Click on the link to open a new window in your browser, where you can explore the Dashboard UI created by the code.

The Dashboard includes a table of data initially populated with all animals from the dataset. Above the table, you'll find a quick filter with several radio buttons, allowing you to filter the animals based on criteria such as water rescue dogs, mountain/wilderness rescue dogs, disaster rescue, and individual tracking dogs.

Beneath the table, you'll find a pie chart and a map. The pie chart displays the percentage and number of animals based on their breed in the table. The map visually represents the location of the selected animal in the table. Both the pie chart and map are dynamically updated whenever the table data changes, providing real-time insights into the dataset.

Now let’s dive into the details of the code to understand how the project is setup.

First, we import all the libraries and packages needed to run the file:

*# Setup the Jupyter version of Dash*

*from jupyter\_dash import JupyterDash*

*# Configure the necessary Python module imports for dashboard components*

*import dash\_leaflet as dl*

*from dash import dcc*

*from dash import html*

*import plotly.express as px*

*from dash import dash\_table*

*from dash.dependencies import Input, Output, State*

*import base64*

*# Configure OS routines*

*import os*

*# Configure the plotting routines*

*import numpy as np*

*import pandas as pd*

*import matplotlib.pyplot as plt*

*from animal\_shelter import AnimalShelter*

Next, initialize an object based on AnimalShelter class from animal\_shelter.py module.

*username = "aacuser"*

*password = "ThiNguyen1412"*

*host = "nv-desktop-services.apporto.com"*

*port = 30301*

*db = "AAC"*

*collection = "animals"*

*shelter = AnimalShelter(username, password, host, port, db, collection)*

Replace username, password, host, port, db and collection with your own credentials and configurations in your machine. The username and password should come from the user account you created in the previous step.

Now create a dataframe used in the table:

*df = pd.DataFrame.from\_records(shelter.read({}))*

*df.drop(columns=['\_id'],inplace=True)*

“shelter.read({})” is the method “read” in the python module used to perform a READ operation in CRUD. This code returns all the documents in the “animals” collection. We put the data in a DataFrame, then drop columns “\_id” because it contains invalid object type of “ObjectID” which will cause the app to crash.

After creating the DataFrame, we initialize a Dash application within this Jupiter Notebook environment: *app = JupyterDash(\_\_name\_\_)*

Next step is building the UI using HTML elements. However, before that, we need to encode the image included in this repo into a base64 format. Feel free to replace the image with your own logo.

*image\_filename = 'Grazioso\_Salvare\_Logo.png'*

*encoded\_image = base64.b64encode(open(image\_filename, 'rb').read()).decode('utf-8')*

Some elements in the HTML such as image or header are straightforward so let’s focus on the main components:

*dcc.RadioItems(*

*id='animal-type-filter',*

*options=[*

*{'label': 'Water Rescue', 'value': 'water'},*

*{'label': 'Mountain or Wilderness Rescue', 'value': 'mountain'},*

*{'label': 'Disaster Rescue or Individual Tracking', 'value': 'disaster'},*

*{'label': 'All Animals', 'value': 'all'}*

*],*

*value='all', # Default value*

*labelStyle={'display': 'inline-block', 'margin-right': '1rem'}*

*)*

This code snippet introduces the creation of a Dash component known as "RadioItems." These radio buttons provide users with the capability to select a single option from among multiple choices. Specifically, we've designed four radio buttons for filtering purposes. Placed at the top of the table in the user interface, these buttons enable users to refine their data view based on specific criteria. Note that the component is initially set with a default value of "all". This decision ensures that upon launching the application, all animals in the dataset are displayed by default. Consequently, this radio button is pre-selected upon app initialization, aligning with the intention to provide users with an immediate and comprehensive view of the dataset.

Following the HTML definition of the radio buttons, you'll find the associated callback function, responsible for updating the table data dynamically based on the selected filter option:  
*@app.callback(*

*Output('datatable-id', 'data'),*

*[Input('animal-type-filter', 'value')]*

*)*

*def on\_radio\_button\_click(selected\_animal\_type):*

*# Query based on Grazioso Salvare's criteria*

*if selected\_animal\_type == 'water':*

*filtered\_data = shelter.read({*

*'breed': {'$in': ['Labrador Retriever Mix', 'Chesapeake Bay Retriever', 'Newfoundland']},*

*'sex\_upon\_outcome': 'Intact Female',*

*'age\_upon\_outcome\_in\_weeks': {'$gte': 26, '$lte': 156}*

*})*

*elif selected\_animal\_type == 'mountain':*

*filtered\_data = shelter.read({*

*'breed': {'$in': ['German Shepherd', 'Alaskan Malamute', 'Old English Sheepdog', 'Siberian Husky', 'Rottweiler']},*

*'sex\_upon\_outcome': 'Intact Male',*

*'age\_upon\_outcome\_in\_weeks': {'$gte': 26, '$lte': 156}*

*})*

*elif selected\_animal\_type == 'disaster':*

*filtered\_data = shelter.read({*

*'breed': {'$in': ['Doberman Pinscher', 'German Shepherd', 'Golden Retriever', 'Bloodhound', 'Rottweiler']},*

*'sex\_upon\_outcome': 'Intact Male',*

*'age\_upon\_outcome\_in\_weeks': {'$gte': 20, '$lte': 300}*

*})*

*else:*

*filtered\_data = shelter.read({})*

*df = pd.DataFrame.from\_records(filtered\_data) # Create df*

*df.drop(columns=['\_id'],inplace=True) # Drop column \_id*

*return df.to\_dict('records') # return filtered list*

The function “on\_radio\_button\_click” will be triggered whenever the value of any input component with ID “animal-type-filter” changes. In other word, when user clicks on one of the radio button, the function is called. Inside the function, based on the selected value of the 'animal-type-filter', different queries are executed on a MongoDB collection named shelter. The queries are designed to filter animals based on Grazioso Salvare's criteria for different types of rescue scenarios (water rescue, mountain rescue, disaster rescue). The queries return new data which is used to update the table.

The next important HTML component is the table:

*dash\_table.DataTable(*

*id='datatable-id',*

*columns=[*

*{"name": i, "id": i, "deletable": False, "selectable": True} for i in df.columns*

*],*

*data=df.to\_dict('records'),*

*editable=False,*

*filter\_action="native",*

*page\_action="native",*

*page\_size=10,*

*page\_current=0,*

*sort\_action="native",*

*column\_selectable=False,*

*row\_selectable="single",*

*row\_deletable=False,*

*selected\_columns=[],*

*selected\_rows=[0],*

*),*

The table is generated using Dash, allowing for interactive data visualization. By default, it's populated with data from the DataFrame established at the beginning of the file. The parameter `editable=False` ensures that users cannot modify the table's fields, maintaining data integrity.

To enhance user experience, we've enabled filtering functionality within the table (`filter\_action="native"`), empowering users to efficiently search for specific keywords within the dataset. However, in cases where the dataset is extensive, displaying all data at once could impact UI performance and readability. To mitigate this, we've implemented pagination (`page\_action="native"`), which divides the dataset into manageable pages.

Pagination starts from the first page (`page\_current=0`), with each page displaying 10 rows (`page\_size=10`). Adjustments to the page size can be made according to specific requirements.

Additionally, the table offers sorting capabilities (`sort\_action="native"`), ensuring data is presented in an organized manner. Columns are not selectable (`column\_selectable=False`), while rows are selectable individually (`row\_selectable="single"`), allowing users to focus on one row at a time. Row deletion functionality is disabled (`row\_deletable=False`), preserving the integrity of the dataset.

Lastly, the initial selected column is set to an empty list (`selected\_columns=[]`), and the initial selected row is set to the first row (`selected\_rows=[0]`). These settings provide users with a clear starting point for interacting with the table, enhancing usability and clarity.

The last component in the Dashboard is a div containing a pie chart (ID “graph-id”) and a map (ID “map-id”):  
*html.Div(className='row',*

*style={'display' : 'flex'},*

*children=[*

*html.Div(*

*id='graph-id',*

*className='col s12 m6',*

*),*

*html.Div(*

*id='map-id',*

*className='col s12 m6',*

*)*

*])*

Below the filter callback function, you can see a callback function for the graph and the map. First, let’s analyze the graph’s function:

*@app.callback(*

*Output('graph-id', "children"),*

*[Input('datatable-id', "derived\_virtual\_data")])*

*def update\_graphs(viewData):*

*if viewData is None:*

*return []*

*df = pd.DataFrame(viewData)*

*breed\_counts = df['breed'].value\_counts()*

*fig = px.pie(names=breed\_counts.index,*

*values=breed\_counts.values,*

*title='Count Based on Breeds')*

*fig.update\_layout(autosize=False,*

*height=500)*

*fig.update\_traces(textposition='inside')*

*fig.update\_layout(uniformtext\_minsize=12, uniformtext\_mode='hide')*

*return [*

*dcc.Graph(*

*figure=fig*

*)*

*]*

The callback is triggered when the table data (datatable-id) changes. This graph serves to illustrate the distribution of each breed within the table, showcasing both the percentage and count of each breed.

If there is no data (i.e., the derived virtual data is None), an empty list is returned, indicating that there is no graph to display. Otherwise, the viewData (derived virtual data from the 'datatable-id' component) is converted into a DataFrame named 'df'. Then, the number of occurrences of each breed is counted using the value\_counts() method, and the result is stored in 'breed\_counts'. A pie chart is created using Plotly Express (px) with breed names as labels and breed counts as values. The size of the chart is adjusted by setting autosize to False and specifying a height of 500 pixels. The chart is configured to enhance readability and user-friendliness: The text labels are positioned inside the pie slices using textposition='inside'; the font size of the labels is adjusted to ensure uniformity and hide labels that are too small to read using uniformtext\_minsize and uniformtext\_mode='hide'. Finally, the pie chart is embedded within a dcc.Graph component and returned as the output of the callback function.  
  
Last but not least, the callback function for the map is located at the end of the file:  
*@app.callback(*

*Output('map-id', "children"),*

*[Input('datatable-id', "derived\_virtual\_data"),*

*Input('datatable-id', "derived\_virtual\_selected\_rows")])*

*def update\_map(viewData, index):*

*if viewData is None or index is None:*

*# No data or no row selected, return an empty map*

*return []*

*dff = pd.DataFrame.from\_dict(viewData)*

*if index is None:*

*row = 0*

*else:*

*row = index[0]*

*# Animal name of the selected row*

*animal\_name = dff.iloc[row, 9]*

*# Check if animal\_name is an empty string or None*

*if not animal\_name:*

*animal\_name = "Unknown" # Set a default value if animal\_name is empty or None*

*# Austin TX is at [30.75,-97.48]*

*return [*

*dl.Map(style={'width': '700px', 'height': '400px'},*

*center=[30.75,-97.48], zoom=10, children=[*

*dl.TileLayer(id="base-layer-id"),*

*# Marker with tool tip and popup*

*dl.Marker(position=[dff.iloc[row,13],dff.iloc[row,14]], # Column 13 and 14 define the grid-coordinates*

*children=[*

*dl.Tooltip(dff.iloc[row,4]), # Defines the breed for the animal*

*dl.Popup([*

*html.H1("Animal Name"),*

*html.P(animal\_name) # Defines animal name*

*])*

*])*

*])*

*]*

This function is invoked whenever there is a change in the table data. Its role is to visualize the geographic location of the currently selected animal from the table on the map.

Inside the function, if either viewData or index is None (i.e., no data or no row is selected), an empty map is returned. Otherwise, the viewData is converted into a DataFrame named 'dff'. The index is used to determine the selected row. If no row is selected (index is None), the first row is selected by default. The animal name of the selected row is retrieved from the DataFrame. If the animal name is empty or None, a default value ("Unknown") is used.

Then, a map is created using Dash Leaflet (dl.Map). It has a predefined style, center, and zoom level.

The map includes a tile layer and a marker representing the location of the selected animal. The marker has a tooltip displaying the breed of the animal and a popup showing the animal's name.

### Tests

In order to test the code, you can add this code snippet after creating the DataFrame:  
*print(len(df.to\_dict(orient='records')))*

*print(df.columns)*

The first line provides a summary of the total number of records present in the dataset, facilitating a comparison with the number of rows in the original CSV file for data validation purposes. The second line lists the names of each column in the dataset, including attributes such as animal\_id, animal\_type, breed, color, and so forth. This ensures all columns are displayed when executing the code.

Next, the filtering functionality is tested. The criteria for filtering are based on breed, sex, and age. To verify the accuracy of the filtered data, the following command can be executed in the MongoDB shell. Make sure to use the right database and collection (“AAC” database and “animals” collection in this case):

For “Water Rescue”:

*db.animals.find({'breed': {'$in': ['Labrador Retriever Mix', 'Chesapeake Bay Retriever', 'Newfoundland']}, 'sex\_upon\_outcome':'Intact Female', 'age\_upon\_outcome\_in\_weeks': {'$gte':26, '$lte':156}}).count()*

For “Mountain or Wilderness Rescue”:

*db.animals.find({'breed': {'$in': ['German Shepherd', 'Alaskan Malamute', 'Old English Sheepdog', 'Siberian Husky', 'Rottweiler']}, 'sex\_upon\_outcome':'Intact Male', 'age\_upon\_outcome\_in\_weeks': {'$gte':26, '$lte':156}}).count()*

For “Disaster Rescue or Individual Tracking”:

*db.animals.find({'breed': {'$in': ['Doberman Pinscher', 'German Shepherd', 'Golden Retriever', 'Bloodhound', 'Rottweiler']}, 'sex\_upon\_outcome':'Intact Male', 'age\_upon\_outcome\_in\_weeks': {'$gte':20, '$lte':300}}).count()*

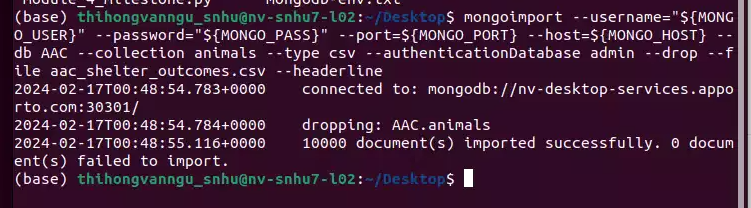
For “All Animals”:

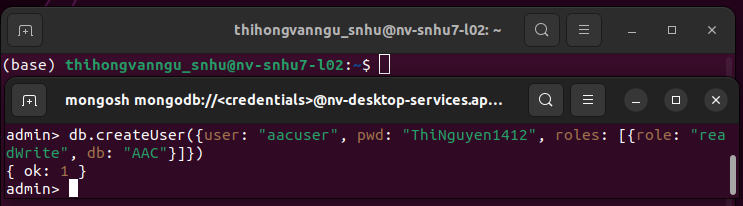
*db.animals.find({}).count()*

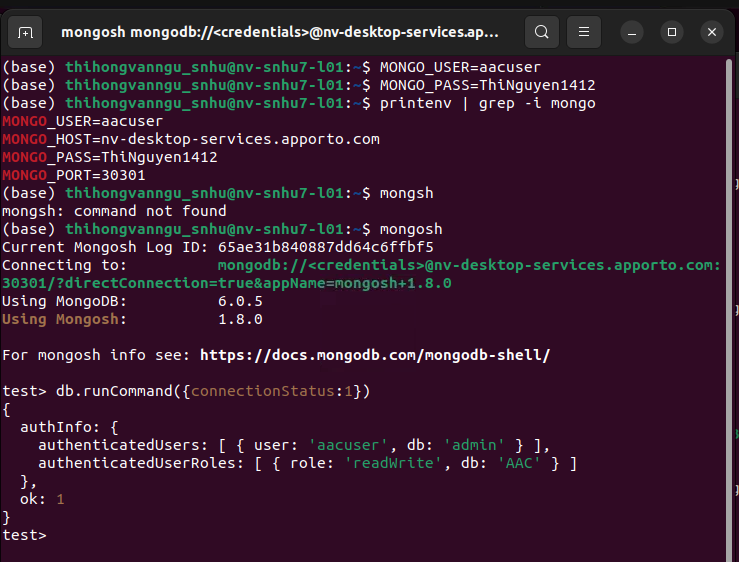
Remove “.count()” to see the details of the list. Upon comparing the data retrieved from the command with the displayed data, you can proceed to further test the functionality of the graph and map. Although accuracy verification can be done visually, it's advisable to include print statements within the callback functions. This allows for comparison with the currently displayed table data, ensuring data consistency and accuracy throughout the testing process.

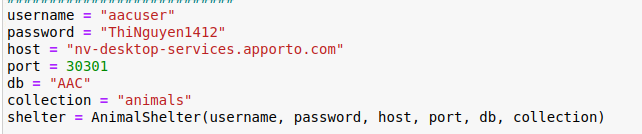
### Screenshots

Below are visual aids in the form of screenshots illustrating key steps, facilitating a more intuitive understanding of the process:

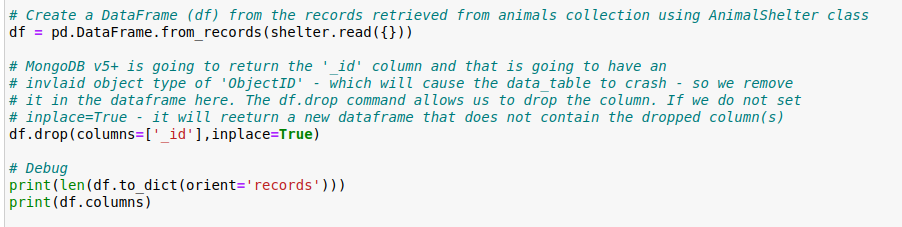
Import target file in mongosh:  


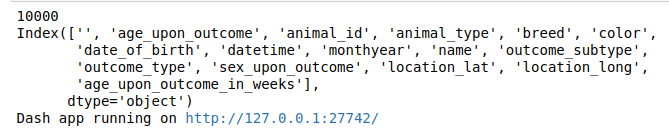
Create a user account:  


Verify the result by login with the created account:  


Initialize AnimalShelter object and store it in shelter variable:  


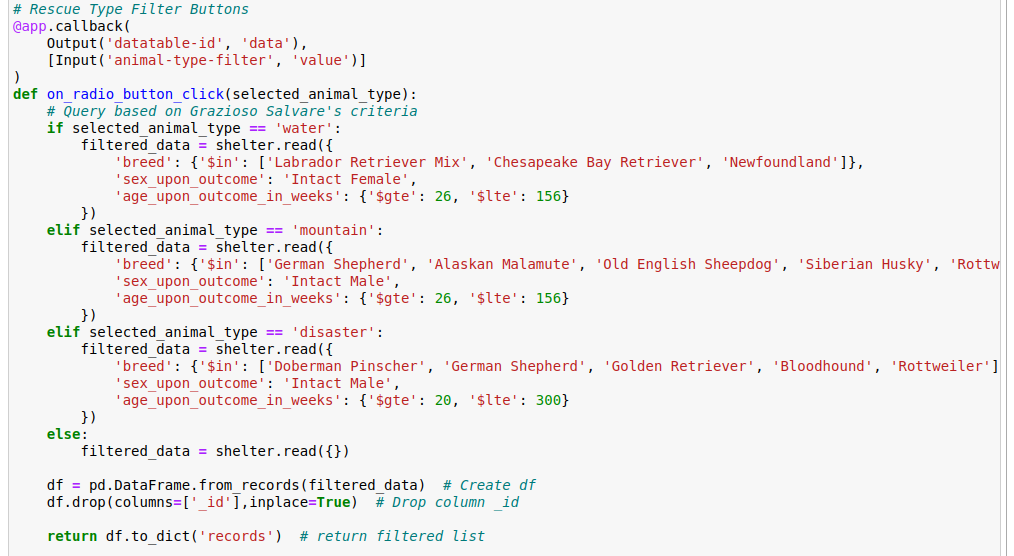
Then, create DataFrame and debug to make sure the data is imported correctly:





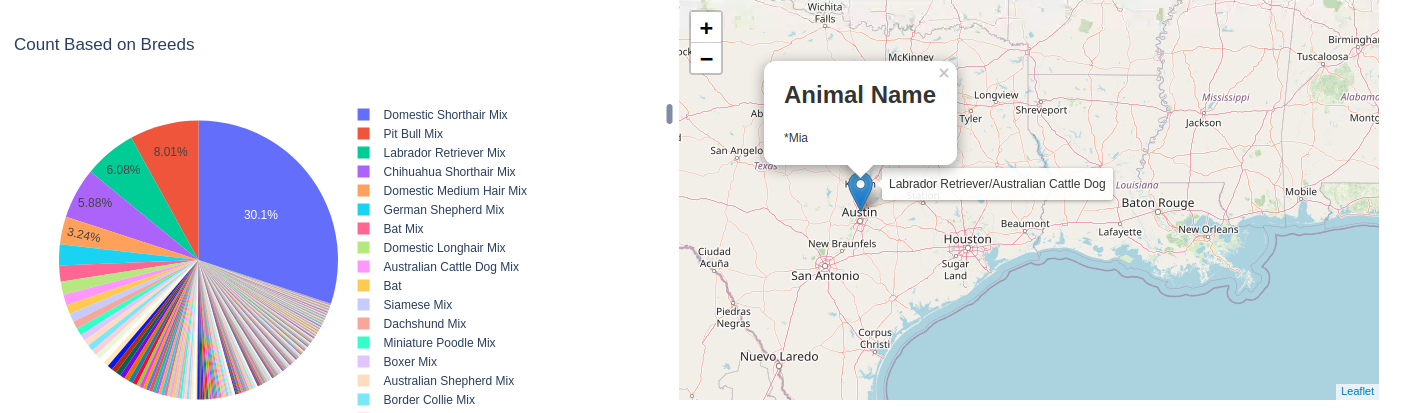
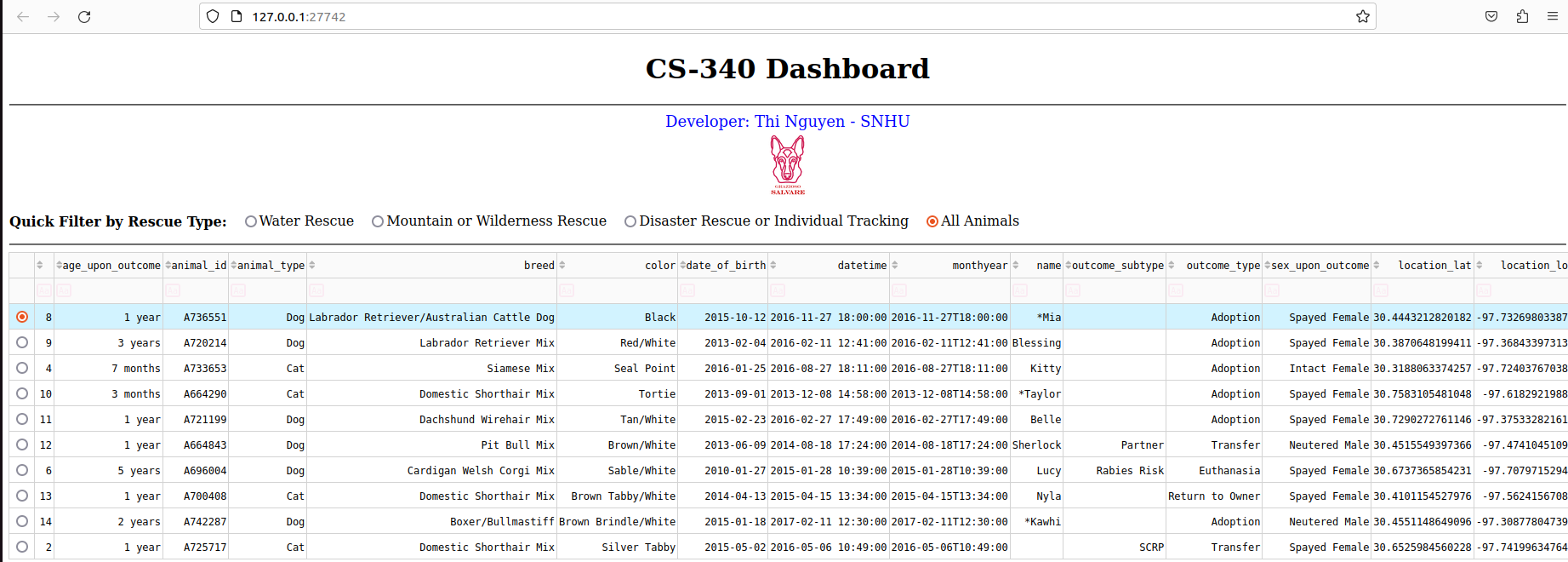
Make sure to encode the image:  

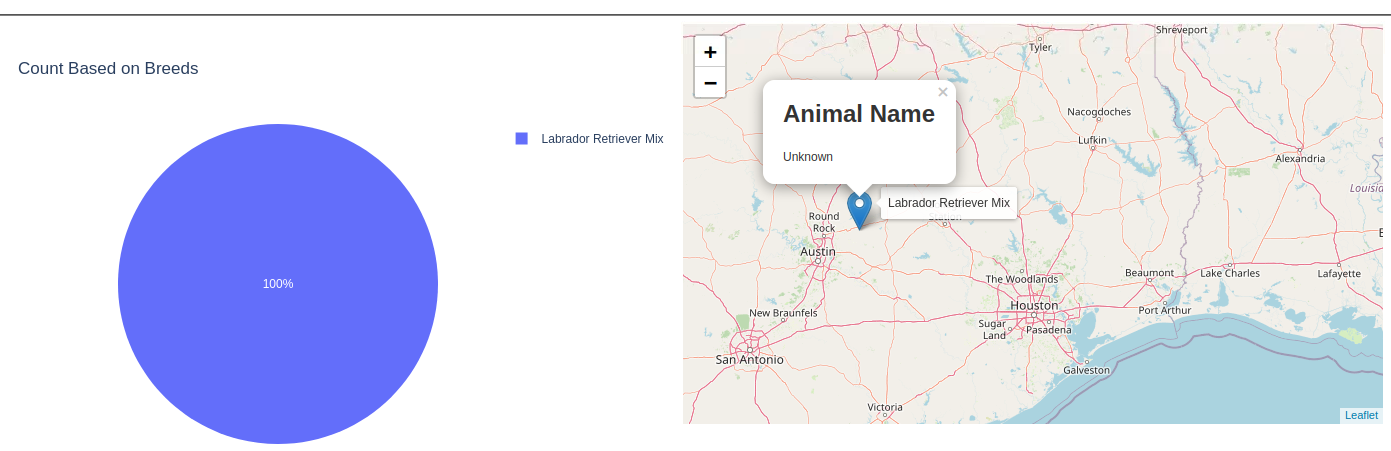
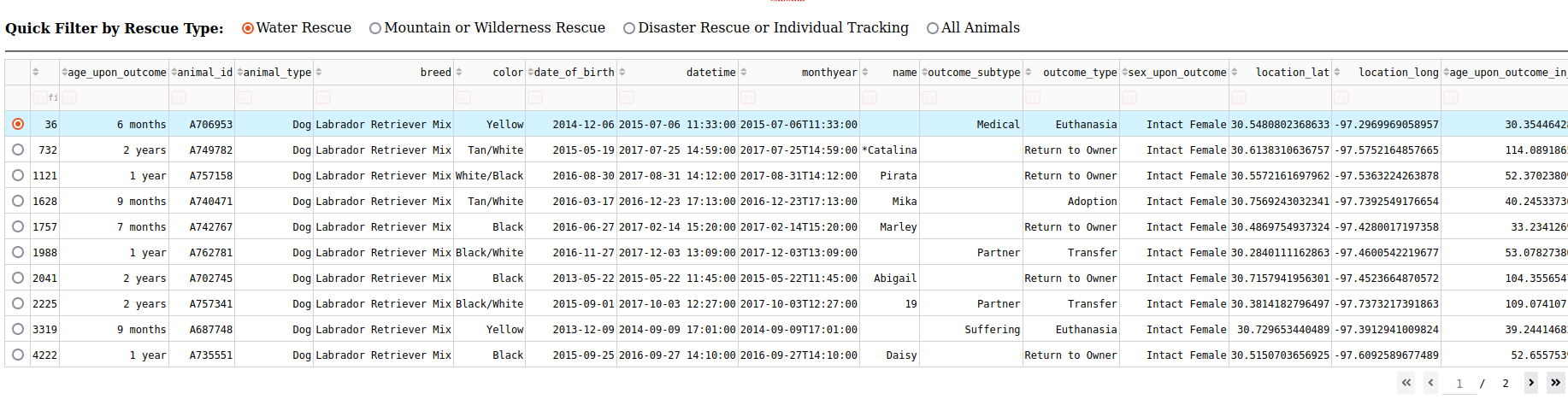

HTML elements for the filter based on rescue type:  

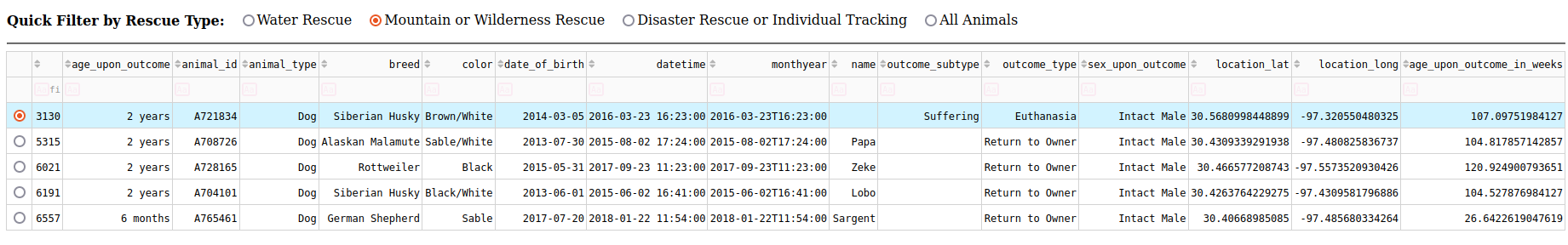

Filter callback function:  


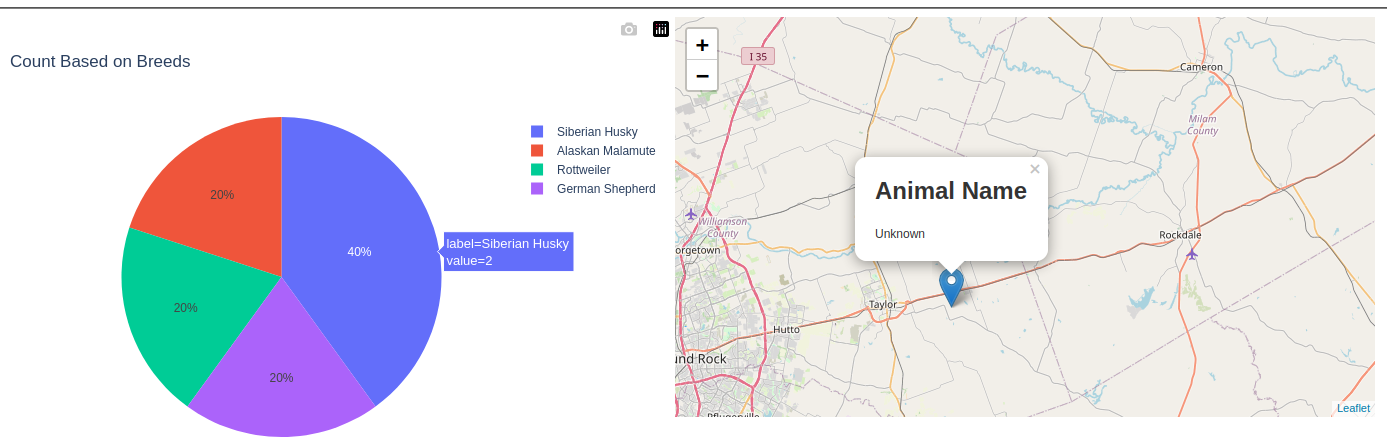
Example of Filter:

All Animals:

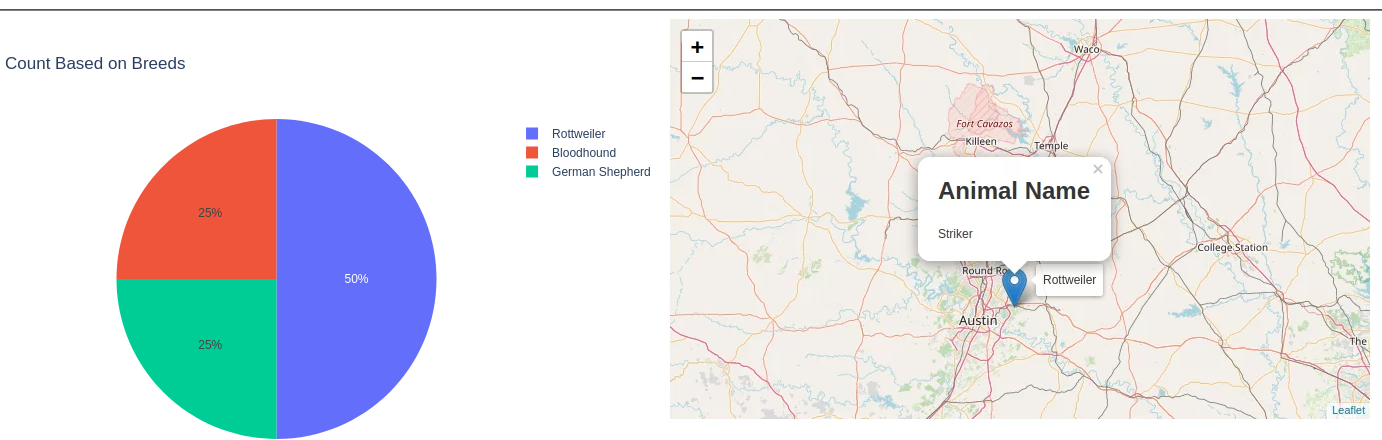
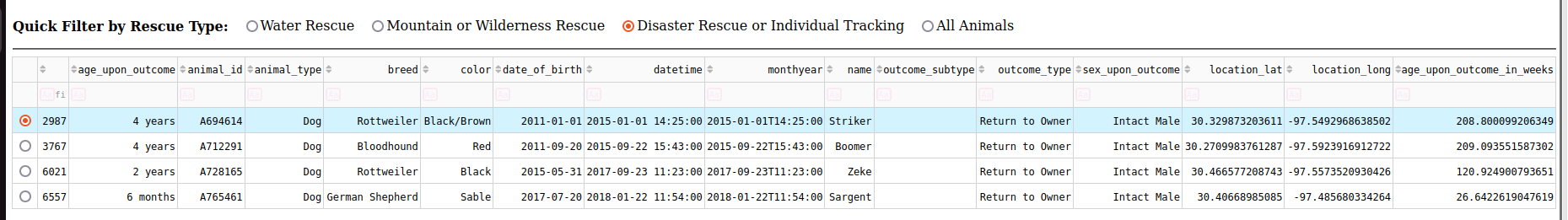


Water Rescue: 

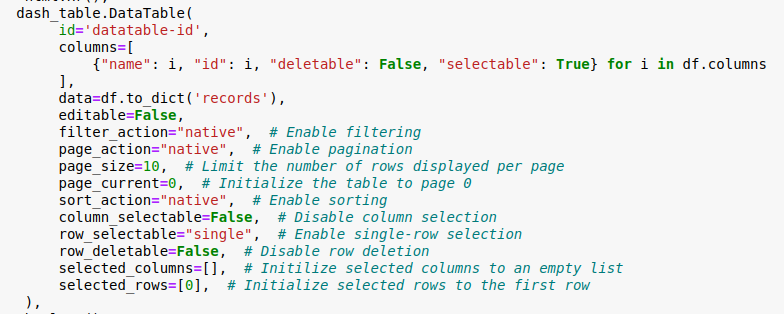
Mountain or Wilderness Rescue:  


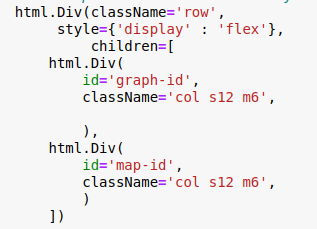


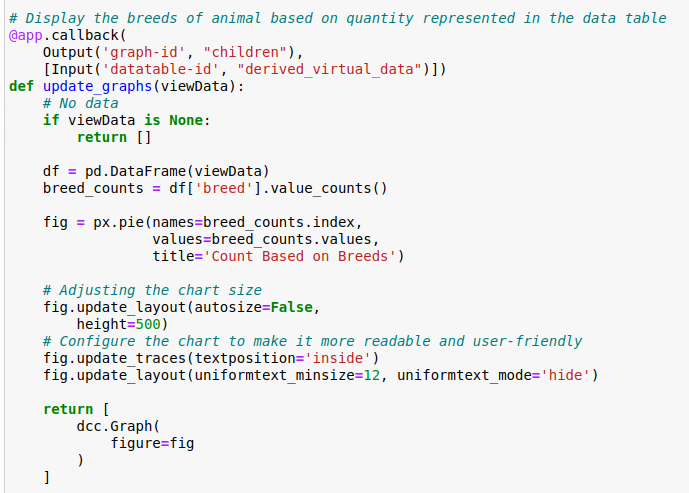
Disaster Rescue or Individual Tracking:

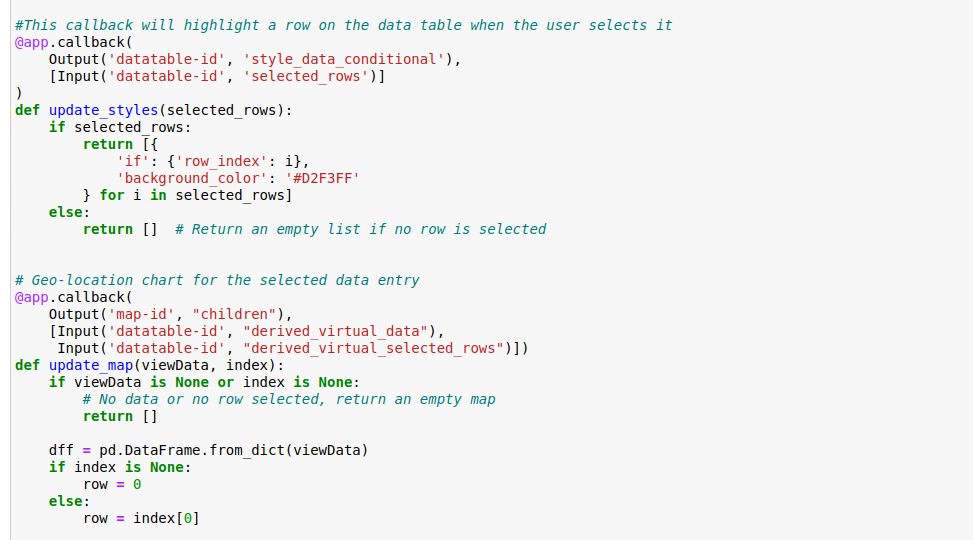


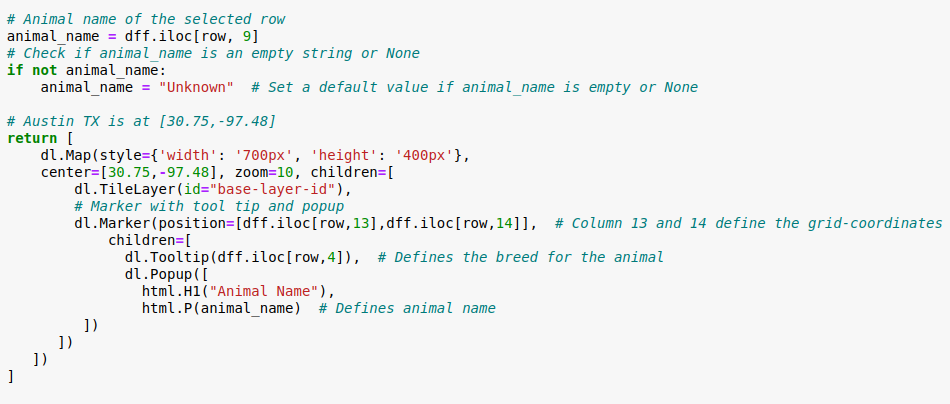
Table’s HTML elements:

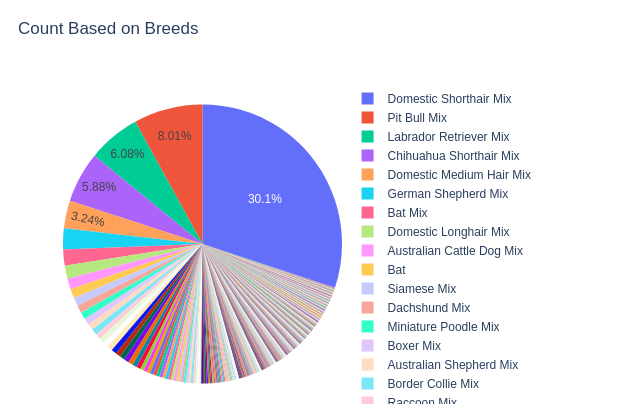


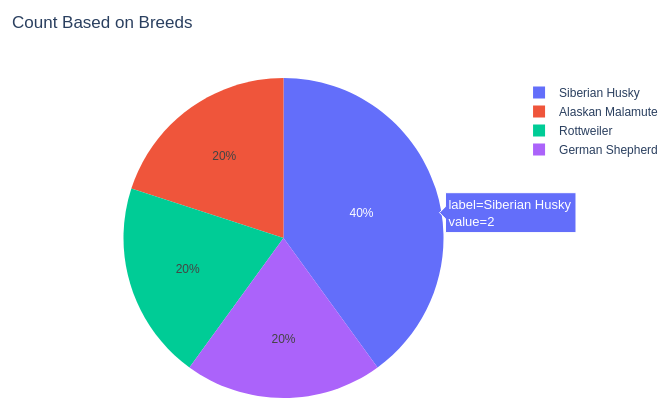
HTML elements for the pie chart and the map:  


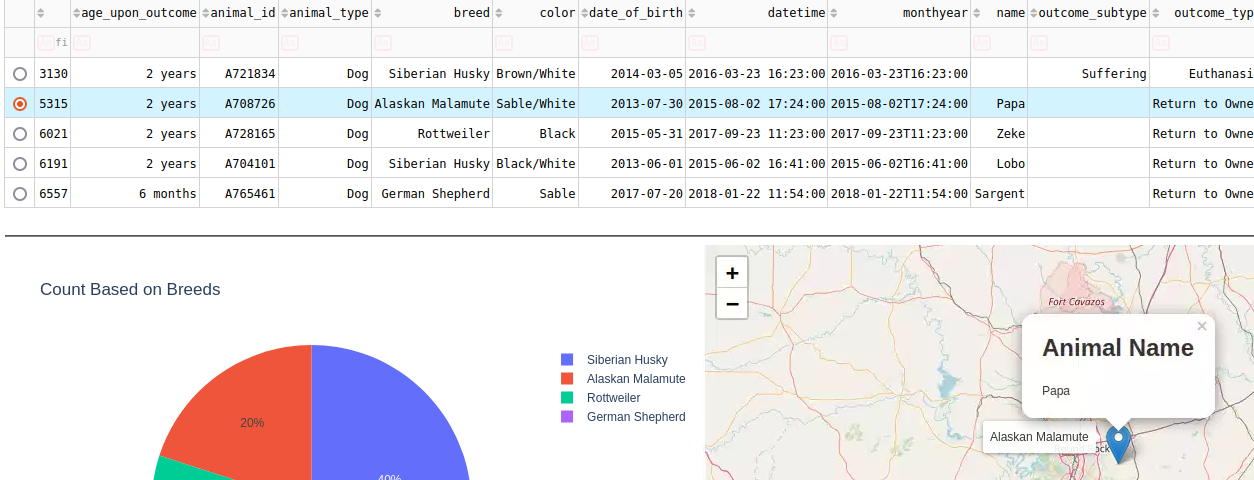
Pie chart and map callback functions:  






Some visual image of the graph and the map:  






## Roadmap/Features (Optional)

The development roadmap for this project involves several key phases aimed at creating a comprehensive MongoDB dashboard for Grazioso Salvare. Initially, the focus lies on establishing a robust database interface logic that facilitates seamless interaction with MongoDB. This entails implementing functionalities for data retrieval, insertion, updating, and deletion, ensuring efficient handling and processing of data within the dashboard environment.

Subsequently, the dashboard's design and layout take precedence, emphasizing the creation of an intuitive and visually appealing interface. The layout is carefully organized to prioritize usability and clarity, with components strategically placed to facilitate easy navigation and interaction for users.

A variety of interactive widgets, including tables, charts (such as pie charts, bar charts), maps, and filters, are developed to provide users with insights into the dataset. These widgets leverage Plotly and Dash libraries to create visually engaging and informative data visualizations, enhanced with features like tooltips, popups, and hover effects for enhanced data exploration and understanding.

Additionally, robust filtering and search functionality are implemented based on various criteria such as breed, age, sex, and rescue proficiency. This empowers users to dynamically filter and search through the dataset to find specific information of interest, further enhancing the dashboard's usability and utility.

**What Makes Our Project Stand Out?**

Our project distinguishes itself through a blend of innovative features and user-centric design, specifically crafted to meet the unique needs of Grazioso Salvare in the rescue-animal training domain. One key aspect setting our project apart is the customized dashboard interface, meticulously designed to provide Grazioso Salvare with an intuitive and efficient means of interacting with their data. The design prioritizes ease of use, ensuring users can access and visualize complex datasets effortlessly.

Moreover, our project incorporates advanced data visualization techniques to present information in a visually compelling and insightful manner. Through interactive charts, graphs, maps, and filters, users can delve deeper into the dataset, gaining valuable insights to inform decision-making processes. Leveraging MongoDB and our custom Python module for CRUD operations, our project offers a robust and reliable database interface. This enables smooth data management and retrieval, empowering Grazioso Salvare to effectively store, query, and analyze their extensive repository of animal data.

Our solution is engineered with scalability and performance in mind, capable of handling large volumes of data without compromising speed or efficiency. Whether it's querying thousands of records or visualizing complex datasets, our dashboard delivers exceptional performance under varying workloads. By making our project open-source and accessible on GitHub, we foster collaboration and innovation within the community. Grazioso Salvare and similar organizations can leverage our codebase to build upon our work, customize features to suit their needs, and contribute back to the project's development.

## Contact

Thi Hong Van Nguyen