# Grazioso Salvare Animal Shelter Database Explorer

## About the Project

This project implements a database explorer which enables non-technical end users to navigate a database. It interfaces with a MongoDB server via a program called “AnimalCRUD.py”. This explorer allows users to search through and filter database entries in an intuitive way, including offering pre-defined search parameters. It features a pie chart which allows users to gain information about the search results and it features a map which allows users to view location data. This project is tailored to a specific database, but with some modification it could be refactored for a different project.

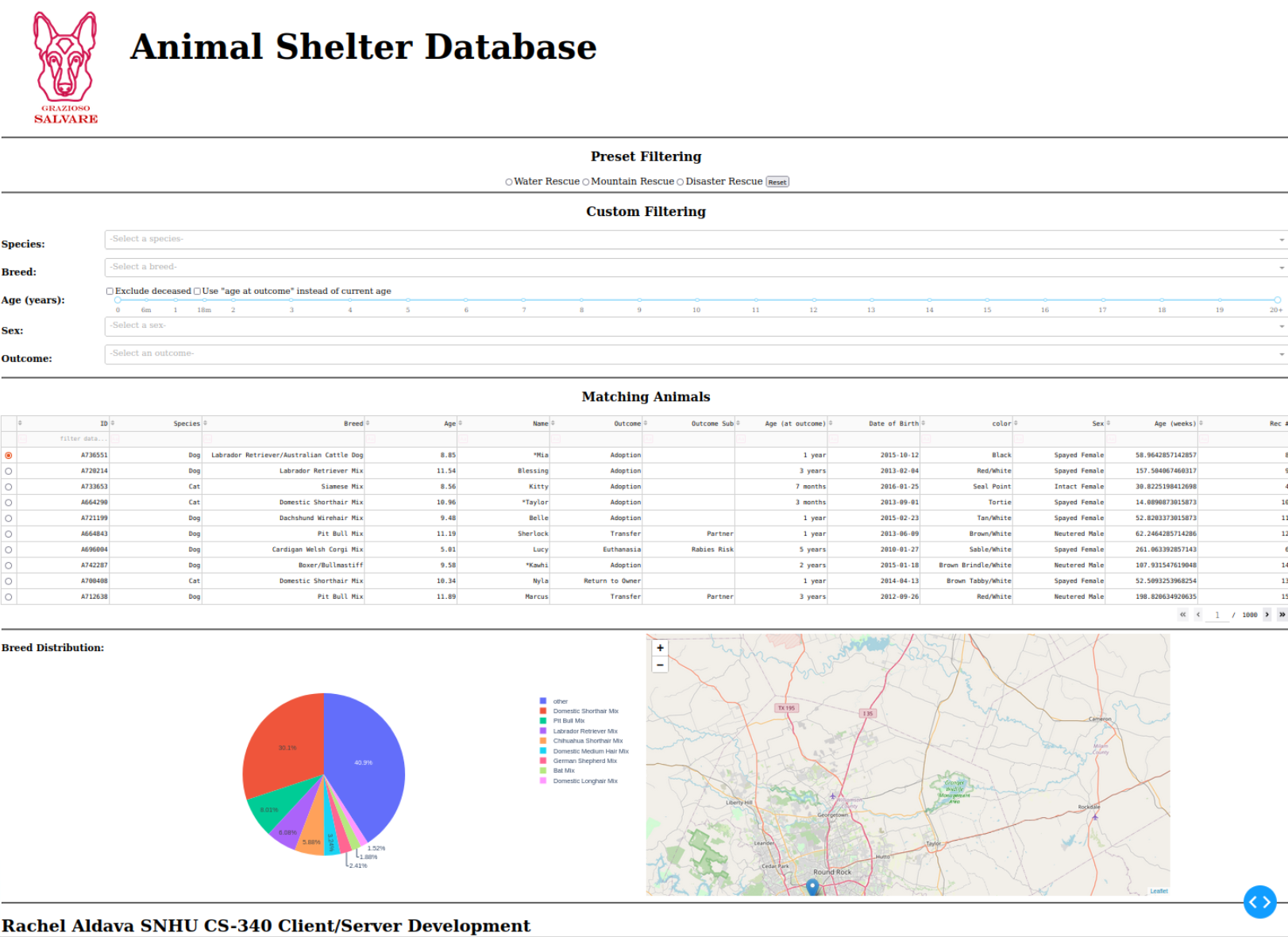
## Motivation

Grazioso Salvare is a canine rescue training organization which needed a program to help them find dogs suitable for training. This project implements a database explorer for the Austin Animal Shelter database. This explorer allows users to filter animals based on their species, breed, age, sex, and “outcome type” (what happened to them at the shelter). Grazsio Salvare has identified the attributes that good rescue dogs will have; users can filter based on dogs suitable for water rescue, mountain rescue, or disaster rescue. The pie chart breaks down the breed distribution of the search results and the map shows the location associated with the animal.

## Getting Started

**Overview:**

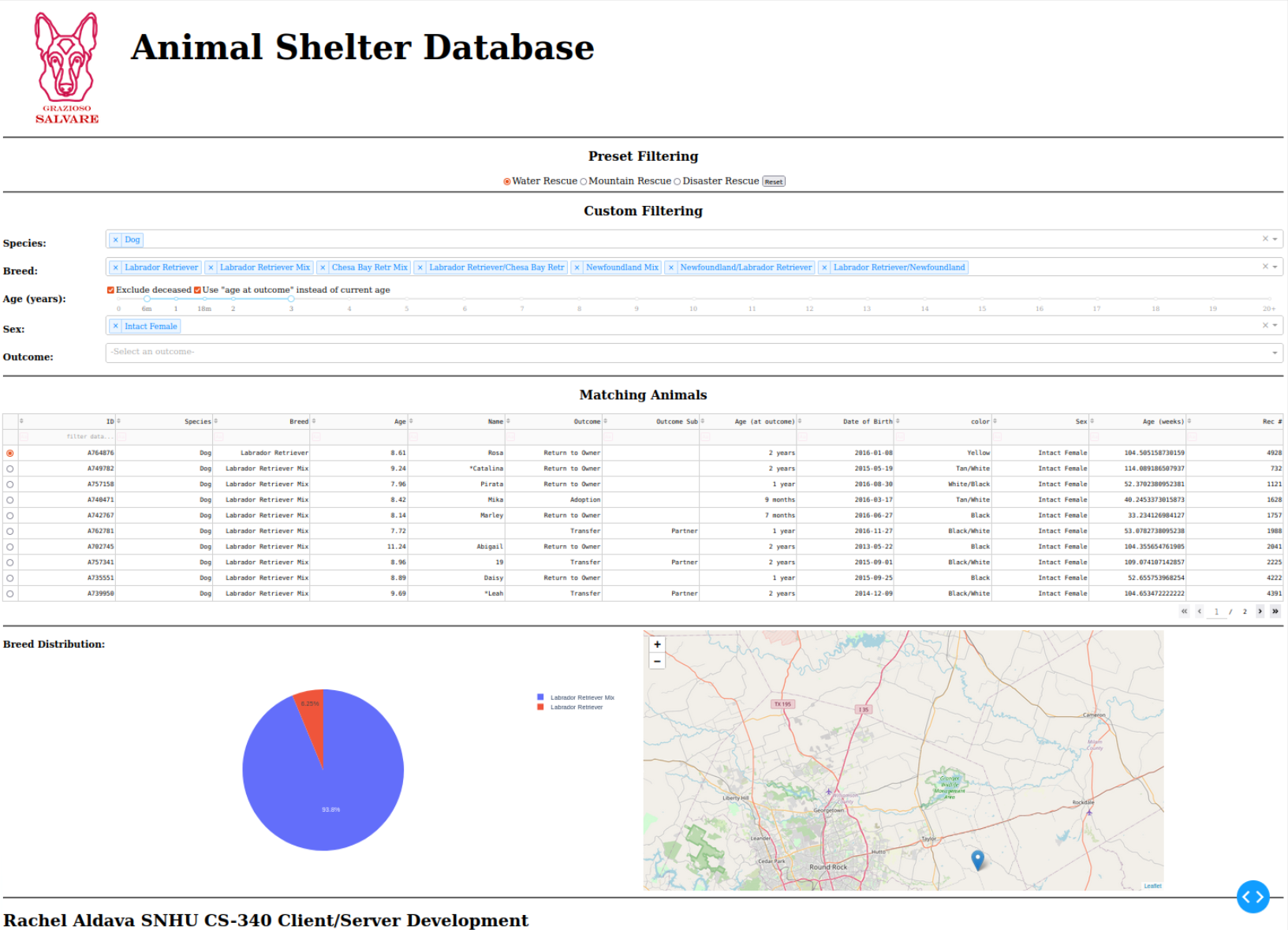
The web page automatically connects with the database. When the program loads on the server it will automatically add or update the database with “age” and “is\_alive” fields for all animals. On loading, users are presented with the following wepage:



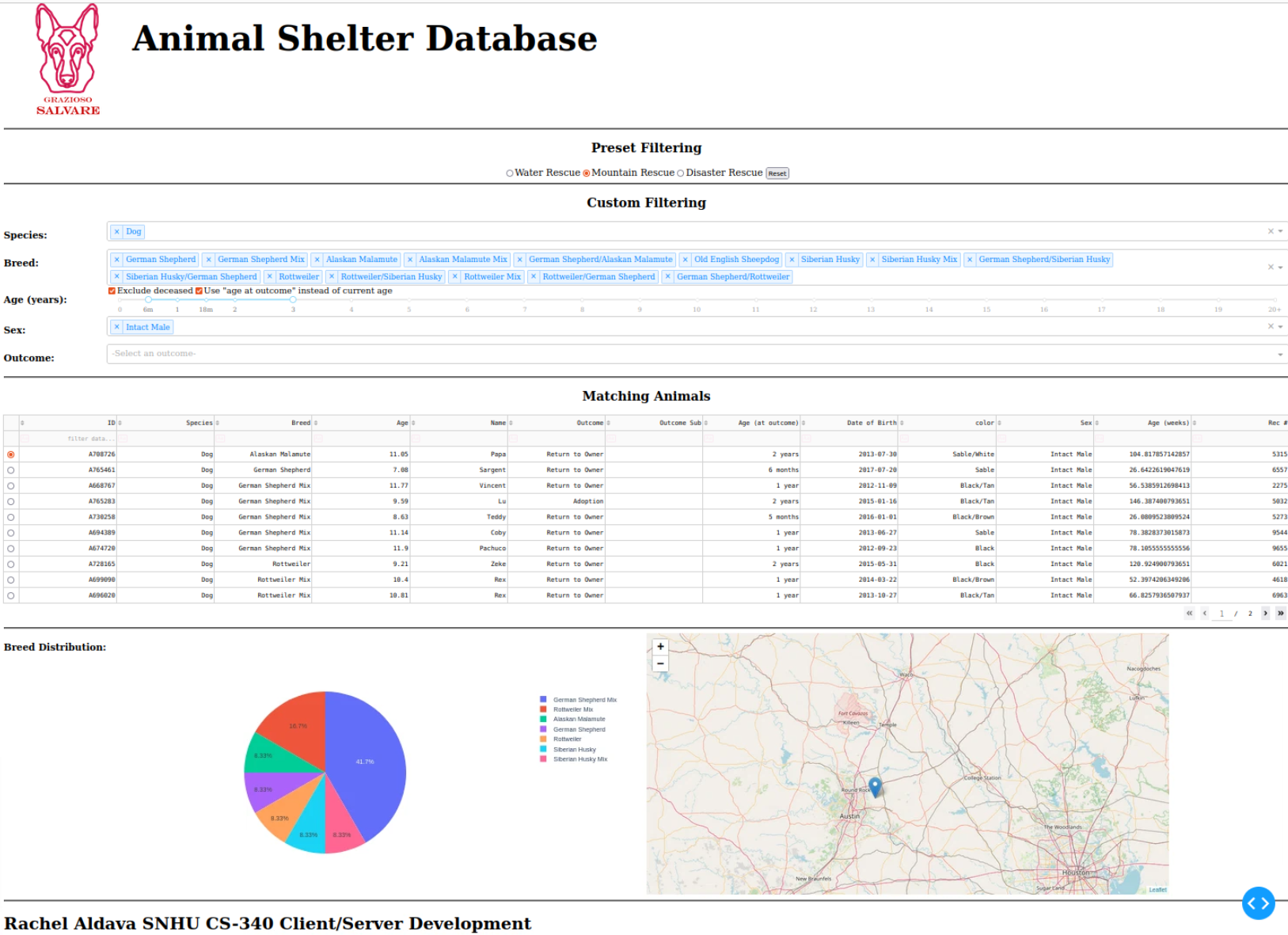
**Filtering:**

Under “Preset Filtering” There are three pre-defined filter buttons along with a reset button. When one of these options is selected, the custom search terms are updated.

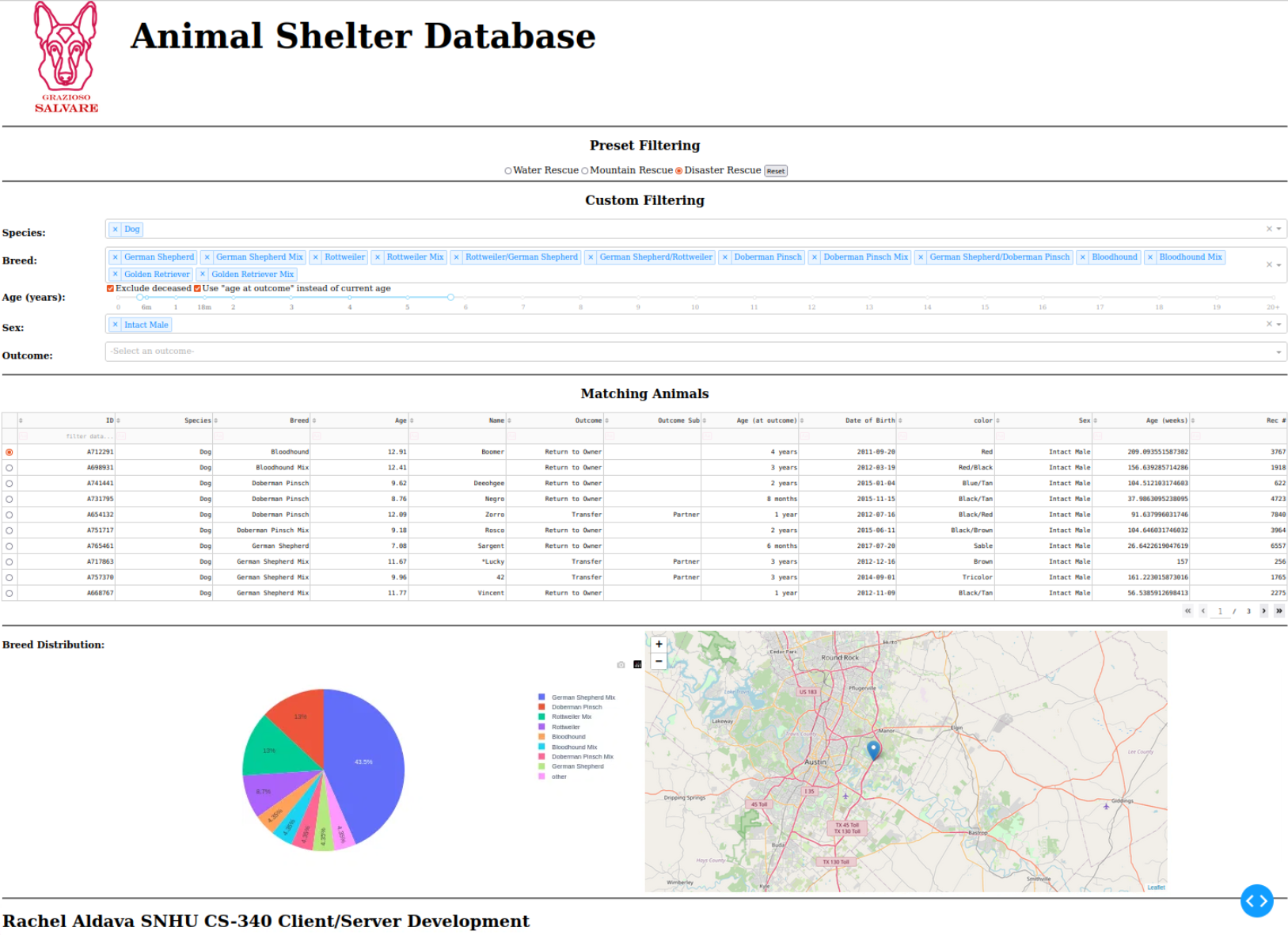
**Water rescue:**



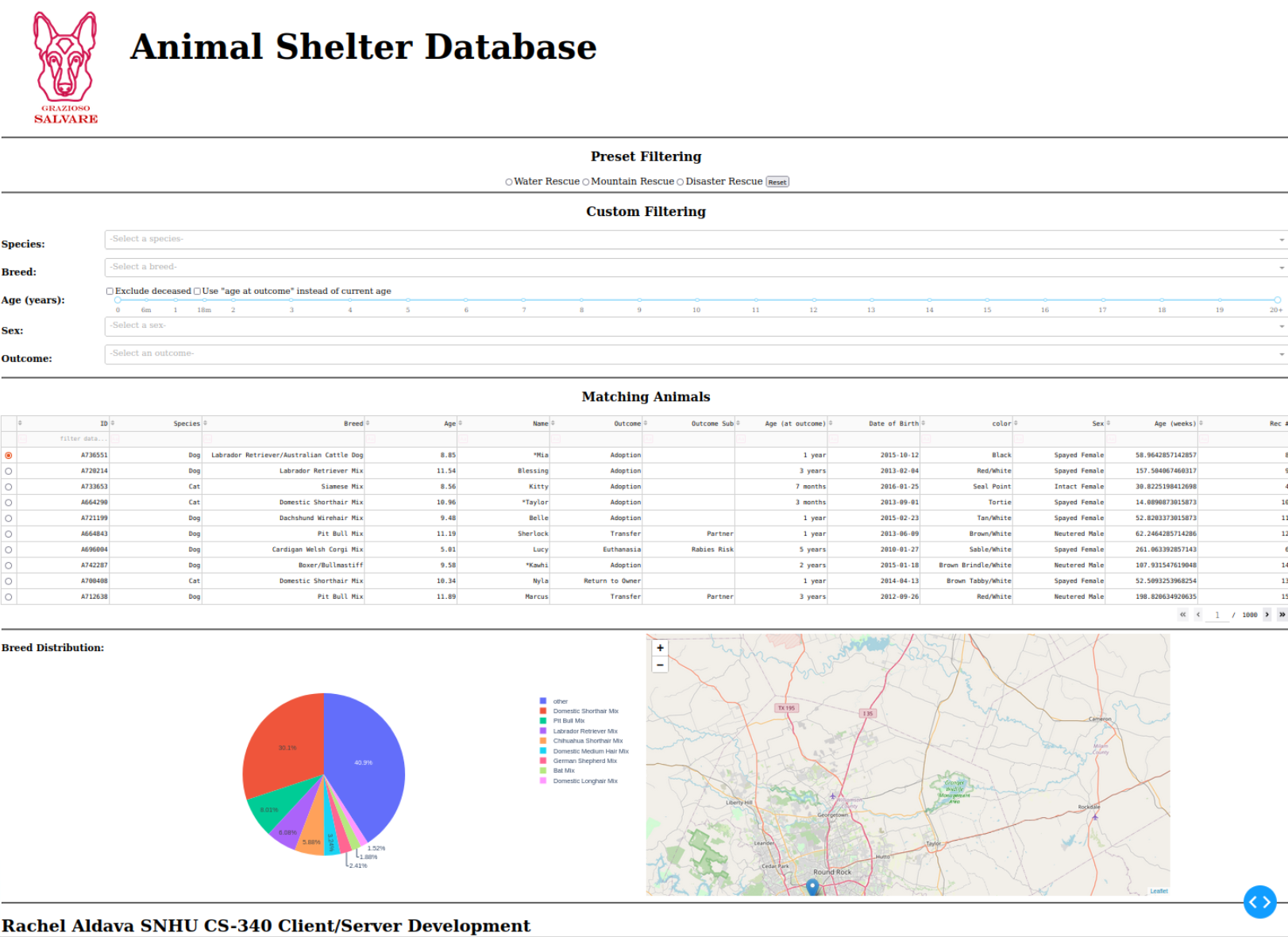
**Mountain Rescue:**



**Disaster Rescue:**

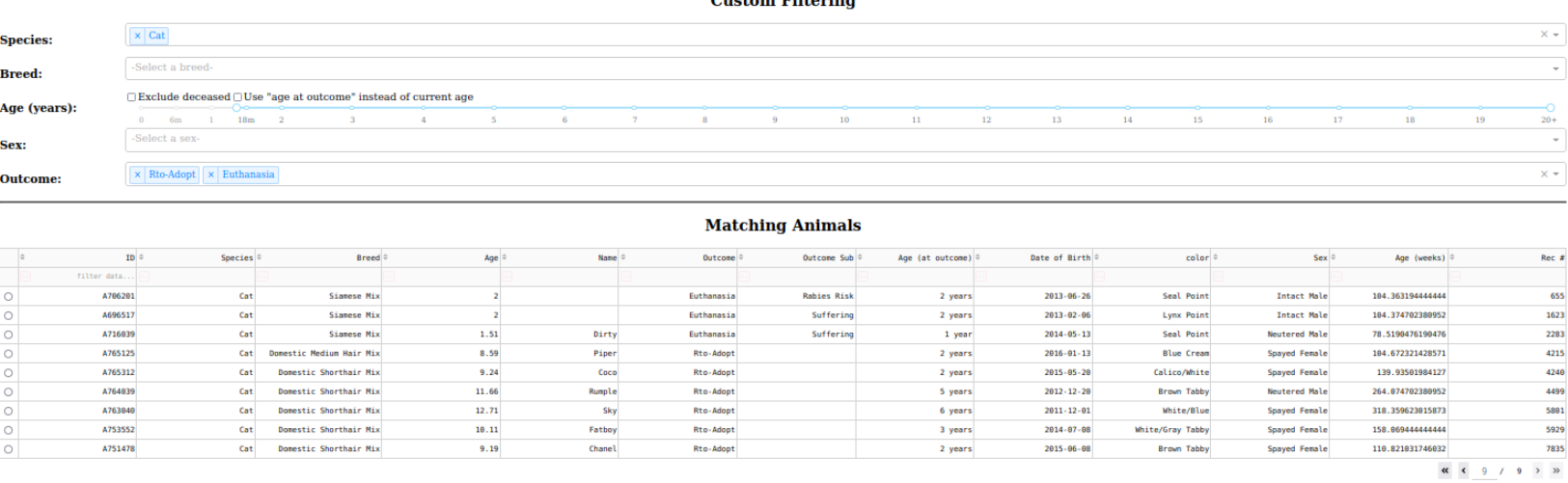
**

**Reset button:**

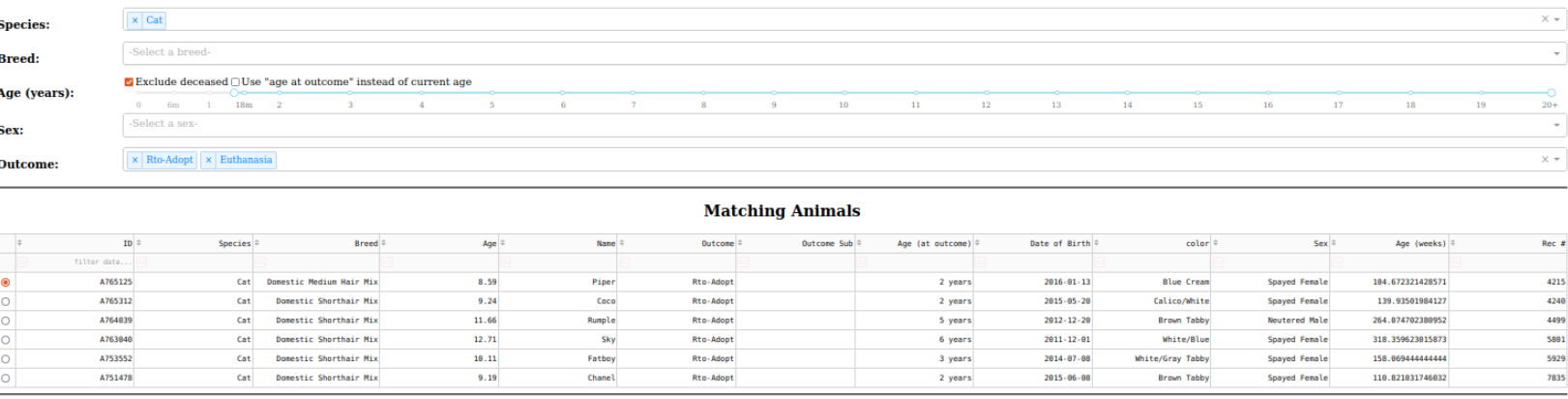


**Custom Filtering:**

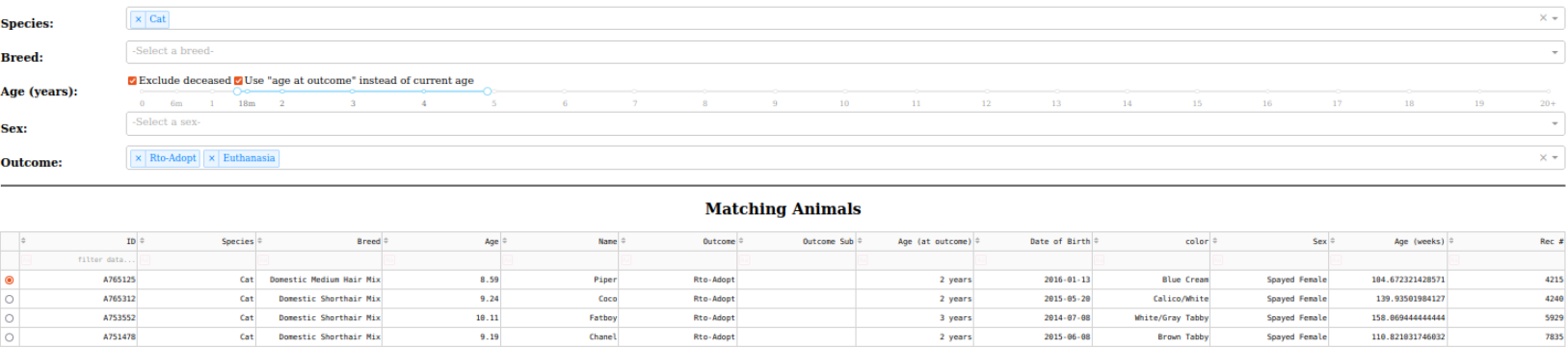
Users can easily add and modify search parameters. For example, this is a search for any cat which had an outcome of “RTO-Adopt” or “Euthanasia” and whose current age is (roughly) above the age of 1 year and 4 months (note: age for deceased animals is based on their final age).

**

If users wish to exclude deceased animals from the database:

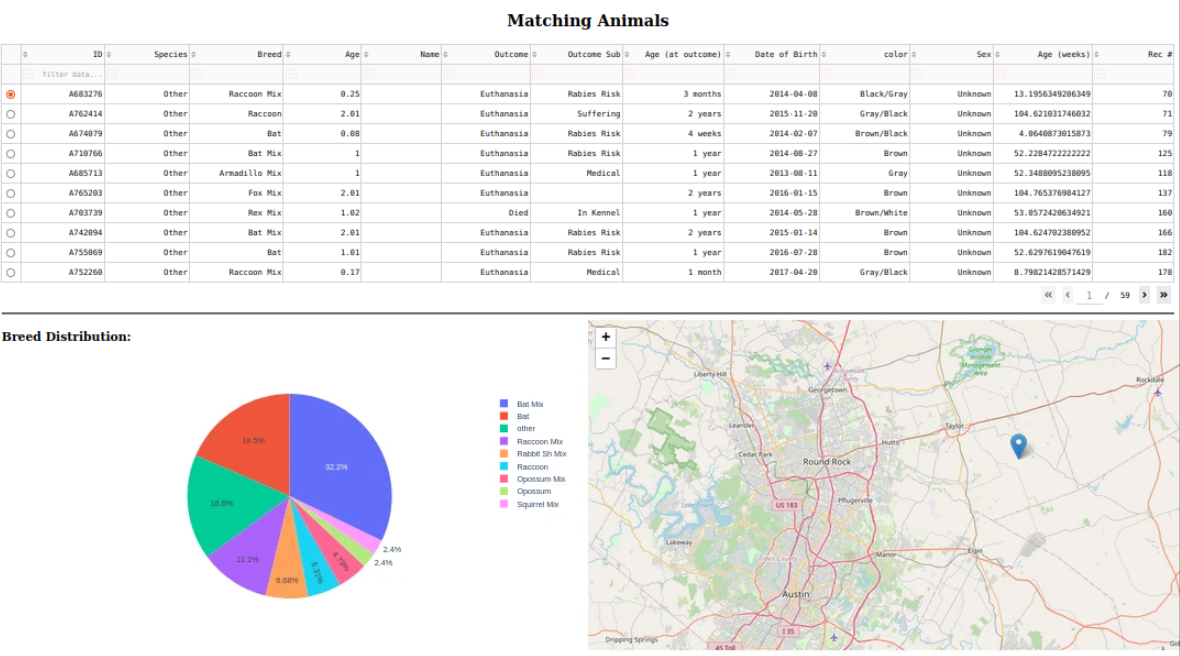
**

If users wish to filter based off of age at outcome, rather than current age (note: maximum age was changed to 4.9 years):

**

**Using graphs:**

The Pie chart automatically updates based off of the “Breed” column. To view locations on a map, users can use the tick-marks in the left-most column:



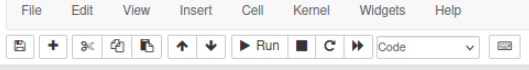


### Installation:

* You will need to have access to a server hosting the Austin Animal Care Mongo database. Installing and setting up this database is beyond the scope of this project. (See AnimalCRUD readme for documentation about connecting to the database)
* Download and install Jupyter: <https://jupyter.org/install>
* Download this repository
* Install dependencies: https://jupyter-contrib-nbextensions.readthedocs.io/en/latest/install.html
  + Install AnimalCRUD.py dependencies (See AnimalCRUD readme)
  + In addition to AnimalCRUD.py, this project has several dependencies:
    - jupyter\_dash
    - dash\_leaflet
    - dash
    - plotly.express
    - dash.dependencies
    - os
    - numpy
    - pandas
    - matplotlib.pyplot
    - datetime

In your system terminal or console, you may run “pip install <dependency>” for example: “pip install pandas”

* Run Jupyter
* Open ProjectTwoDashboard.ipynb
* Press the “run” button until you see an output





### Technologies used:

* MongoDB

MongoDB is a database technology that allows for a large number of documents to be stored. Its inclusion in this project was a condition set by the client. Its high performance allows for speedy search results.

* PyMongo

PyMongo is used extensively in AnimalCRUD.py; it allows for queries to be passed from Python code to the Mongo database. The query structure of PyMongo allows for a search query to be built piecemeal then executed at a later date. This capability was leveraged with the preset buttons. Multiple search parameters were set at the same time, and instead of triggering a new query for each time one parameter was updated, an entire query was constructed using all search parameters and THEN executed.

* Jupyter

Jupyter is the python IDE which was used. There is no special reason for this choice; Anything that can run python code should be suitable for this task.

* Pandas

Pandas is a python package which allows for data manipulation. It was used in several intermediate steps in order to perform appropriate data transformations, such as enabling the pie chart to show “other” for less common breeds.

* Dash

Dash is a framework which allows for web pages to be built in python without writing a large amount of code. Dash has pre-built components which can be pieced together to form a larger web page. Since, at time of writing, the developer’s only experience with web pages was constructing an HTML home page for her local girl scout troop back in the mid 90s, Dash provided a very simple way to leverage existing python knowledge when constructing a front-end.

### Development and Reproduction:

The development of AnimalCRUD.py and setting up the mongo database have been documented in the AnimalCRUD readme. This section will discuss the development of the python script. In the Jupyter notebook, the code is divided into three sections: Importing, setup, and the main body of the project.

**Importing:**

# 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  
  
  
# import CRUD  
from AnimalCRUD import AnimalShelter  
import datetime

The setup section is very typical of a python script. It provides instructions for python to load the various pieces of code that we need. The only notable addition is the line from AnimalCRUD import AnimalShelter Which instructs python to load AnimalCRUD.py. This file needs to be placed in the same location as ProjectTwoDashboard.ipynb. If you wish to have these files in separate locations, then in this line you must specify the location of AnimalCRUD.py.

**Setup:**

###########################  
# Data Manipulation / Model  
###########################  
  
# Connect to database via CRUD Module  
username = "aacuser"  
password = "aacpassword"  
shelter = AnimalShelter(username, password)  
  
# Converts a string with a date in the format of "2015-11-29".  
# This is the format used for the ACC database in documents with the key "date\_of\_birth".  
# Other formats are not supported.  
def GetAge(datetime\_string):  
 try:  
 current\_time = datetime.datetime.now() # Get the current time  
 date = datetime.datetime.strptime(datetime\_string, "%Y-%m-%d")# Convert the supplied string into a date  
 year\_old = current\_time.year - date.year  
 month\_old = (current\_time.month - date.month) / 12 # The difference in months expressed in terms of a year  
 day\_old = (current\_time.day - date.day) / 365 # The difference in days expressed in terms of a year  
 return(round(year\_old + month\_old + day\_old, 2))  
 except:  
 return 0  
  
# This will update documents in the database with their current age and whether or not they are deceased.  
# The intent is for this query to be run on a daily basis. For an actual implementation, this process should be  
# run as a re-occuring script rather than featured here.  
# Calculating age infrequently, rather than individually at runtime, should slightly improve performance.  
def update\_database\_age\_and\_living():  
 deceased\_outcomes = ["Euthanasia", "Died", "Disposal"]  
 weeks\_per\_year = 52.1429  
 for doc in shelter.read({}):# For every document  
 dob = doc['date\_of\_birth']  
 id = doc['\_id']  
 # Boolean of whether the "outcome type" of the document is within the above list  
 is\_deceased = (doc['outcome\_type'] in deceased\_outcomes)  
  
 if is\_deceased:  
 if doc['is\_deceased']:  
 pass# Do nothing if the animal is already marked as deceased  
 else:  
 # Call GetAge() to calculate the age at time of death and create or update the "age field"  
 # and create or update the is\_deceased field  
 shelter.update\_one({"\_id" : id}, {"age" : (round(doc['age\_upon\_outcome\_in\_weeks'] / weeks\_per\_year, 2)), "is\_deceased" : True})  
 else:  
 # Else the animal is alive; udate age and ensure is\_deceased is false  
 shelter.update\_one({"\_id" : id}, {"age" : GetAge(dob), "is\_deceased" : False})  
update\_database\_age\_and\_living()  
  
# Load Grazioso Salvare’s logo  
image\_filename = 'Grazioso Salvare Logo.png' # replace with your own image  
encoded\_image = base64.b64encode(open(image\_filename, 'rb').read())

This section is very important in order for the test of the code to function. First, we instantiate the object by supplying a valid username and password (for more details, see the AnimalCRUD readme).

Next, the original database did not feature age or living status, but this information could plausibly be inferred. We were given permission to modify the database and in order to demonstrate our ability to write data to the database we were encouraged to modify it. Two methods are defined: GetAge() and update\_database\_age\_and\_living(). GetAge() will return a float value that is the difference in years between a supplied date and the current date. update\_database\_age\_and\_living() will interate through the entire database: If the animal has an outcome which indicates that it is deceased, then that information, along with the age of death, will be calculated. Otherwise, the current age and living status will be updated. **These two methods are the most computationally expensive of this project** and are intended to be run very infrequently; age need not be updated multiple times per day, and perhaps it should be calculated weekly. Currently, they run every time the python script is executed. The reason we are spending this computation time is because it will improve the speed of our query outputs.

Finally, we load the company logo into memory so that we can display it later.

**Main body:**

**Header layout:**

#########################  
# Dashboard Layout / View  
#########################  
app = JupyterDash(\_\_name\_\_)

app.layout = html.Div([  
 ################ HEADER ################  
 html.Div(className='row', style={'display': 'flex', "width": "100%"}, children=[  
 html.Img(  
 id='customer-image',  
 src='data:image/png;base64,{}'.format(encoded\_image.decode()),  
 alt='customer image',  
 style={'height': '10%', 'width': '10%'}),  
  
 html.Div(style={'font-size': '32px'}, children=[  
 html.Center(html.B(html.H1('Animal Shelter Database')))  
 ])  
 ]),

The entire format of the page is contained within the app.layout value. An “html.div” is a way of sub-dividing the code into sections. They can contain style data such as text size, color, alignment, and the size of the sub-divided box. App.layout contains a single html.div, but that html.div has many html.div objects within it. In the header, we define one of these sub html.div, specify it will be in the format of a row, we insert the image and text, and specify the size of both.

**Preset Filtering:**

################ Div line ################  
html.Hr(),  
################ Preset filtering ################  
html.Center(html.B(html.H2('Preset Filtering'))),  
  
html.Div(className='row', style={'display': 'flex', 'font-size': '18px', "justifyContent": "center"}, children=[  
 # Radio items  
 html.Div([  
 dcc.RadioItems(['Water Rescue', 'Mountain Rescue', 'Disaster Rescue'], inline=True,  
 id='filter\_preset\_radio'),  
 ]),  
 # reset button  
 html.Div(style={"margin-left": "5px"}, children=[  
 html.Button('Reset', id='filter\_preset\_reset'),  
 ]),  
 dcc.Store(id='preset\_selection', data=None),  
]),

In the preset filtering section, we write the title of the section and define an html.div to contain the components we will place inside. We create the radio items, specify what choices the user has, and associate “filter\_preset\_radio” as the ID for the object. Similarly, we create a button named “filter\_preset\_reset”, and lastly we create a “dcc.Store” object named “preset\_selection”. The dcc.Store component is essentially just a container that holds a variable; it is not visible in the web page.

**Custom Filtering:**

################ Div line ################  
html.Hr(), # Div line  
html.Div([  
 ################ Custom Filtering ################  
  
 html.Center(html.B(html.H2('Custom Filtering'))),  
 dcc.Store(id='filter', data={}),  
 # Species Filter  
 html.Div(className='row', style={'display': 'flex'}, children=[  
 # Text  
 html.Div([  
 html.H2('Species:', style={'font-size': '18px'})  
 ], style={"width": "8%"}),  
 # Dropdown  
 html.Div([  
 dcc.Dropdown(  
 id="filter\_species",  
 options=[{"label": species, "value": species} for species in df.animal\_type.unique().tolist()],  
 placeholder="-Select a species-",  
 multi=True,  
 value=df.animal\_type.unique().tolist())  
 ], style={"width": "92%"}),  
 ]),  
  
 # Breed Filter  
 html.Div(className='row', style={'display': 'flex'}, children=[  
 # Text  
 html.Div([  
 html.H2('Breed:', style={'font-size': '18px'})  
 ], style={"width": "8%"}),  
 # Dropdown  
 html.Div([  
 dcc.Dropdown(  
 id="filter\_breed",  
 options=[{"label": breed, "value": breed} for breed in df.breed.unique().tolist()],  
 placeholder="-Select a breed-",  
 multi=True)  
 ], style={"width": "92%"}),  
 ]),  
  
 # Age Filter  
 # NOTE: The user sees age in years, but calculations are performed in terms of weeks per client request  
 html.Div(className='row', style={'display': 'flex'}, children=[  
 # Text  
 html.Div([  
 html.H2('Age (years):', style={'font-size': '18px'})  
 ], style={"width": "8%"}),  
 # Dropdown  
 html.Div([  
 dcc.Checklist(  
 ['Exclude deceased', 'Use "age at outcome" instead of current age'],  
 ['Use "age at outcome" instead of current age'],  
 id="filter\_age\_options",  
 ),  
 dcc.RangeSlider(  
 id="filter\_age",  
 min=0,  
 max=1043,  
 step=1,  
 marks={0: '0', 26: '6m', 52: '1', 78: '18m', 104: '2', 156: '3', 209: '4', 261: '5',  
 313: '6', 365: '7', 417: '8', 469: '9', 521: '10', 574: '11', 626: '12', 678: '13',  
 730: '14', 782: '15', 834: '16', 886: '17', 939: '18', 991: '19', 1042: '20+'  
 },  
 value=[0, 1043])  
 ], style={"width": "92%"}),  
 ]),  
  
 # Sex Filter  
 html.Div(className='row', style={'display': 'flex'}, children=[  
 # Text  
 html.Div([  
 html.H2('Sex:', style={'font-size': '18px'})  
 ], style={"width": "8%"}),  
 # Dropdown  
 html.Div([  
 dcc.Dropdown(  
 id="filter\_sex",  
 options=[{"label": sex, "value": sex} for sex in df.sex\_upon\_outcome.unique().tolist()],  
 placeholder="-Select a sex-",  
 multi=True,  
 value=df.sex\_upon\_outcome.unique().tolist())  
 ], style={"width": "92%"}),  
 ]),  
  
 # Outcome Filter  
 html.Div(className='row', style={'display': 'flex'}, children=[  
 # Text  
 html.Div([  
 html.H2('Outcome:', style={'font-size': '18px'})  
 ], style={"width": "8%"}),  
 # Dropdown  
 html.Div([  
 dcc.Dropdown(  
 id="filter\_outcome",  
 options=[{"label": outcome, "value": outcome} for outcome in df.outcome\_type.unique().tolist()],  
 placeholder="-Select an outcome-",  
 multi=True,  
 value=df.outcome\_type.unique().tolist())  
 ], style={"width": "92%"}),  
 ]),  
]),

Custom filtering is a fairly large section, but slightly repetitive. At the top we define a store which we will later use to house our search parameter. Most elements are dropdown menus with names such as “filter\_outcome”. Users can select these values and they will be stored as a value inside of the object. The age filter contains slightly more complexity: There is a checklist which functions very similar to the radio component described previously; the only difference is that the string values will be stored in a list. The other component is a RangeSlider, which allows users to specify between two numbers. These numbers are stored as a list [x,y] where x is the lower bound and y is the upper bound. The values and steps are measured in weeks, but the markings are displayed in years. For example, 991 weeks is very close to 19 years. The reason this decision was made is because the client wishes for pre-defined calculations to be performed using weeks.

**Table layout:**

################ Div line ################  
html.Hr(), # Div line  
################ Table ################  
html.Center(html.B(html.H2("Matching Animals"))),  
dash\_table.DataTable(  
 id='datatable-id',  
 columns=[  
 {  
 "name": "ID",  
 "id": "animal\_id",  
 "deletable": False,  
 "selectable": True  
 },  
 {  
 "name": "Species",  
 "id": "animal\_type",  
 "deletable": False,  
 "selectable": True  
 },  
 {  
 "name": "Breed",  
 "id": "breed",  
 "deletable": False,  
 "selectable": True  
 },  
 {  
 "name": "Age",  
 "id": "age",  
 "deletable": False,  
 "selectable": True  
 },  
 {  
 "name": "Name",  
 "id": "name",  
 "deletable": False,  
 "selectable": True  
 },  
 {  
 "name": "Outcome",  
 "id": "outcome\_type",  
 "deletable": False,  
 "selectable": True  
 },  
 {  
 "name": "Outcome Sub",  
 "id": "outcome\_subtype",  
 "deletable": False,  
 "selectable": True  
 },  
 {  
 "name": "Age (at outcome)",  
 "id": "age\_upon\_outcome",  
 "deletable": False,  
 "selectable": True  
 },  
 {  
 "name": "Date of Birth",  
 "id": "date\_of\_birth",  
 "deletable": False,  
 "selectable": True  
 },  
 {  
 "name": "color",  
 "id": "color",  
 "deletable": False,  
 "selectable": True  
 },  
 {  
 "name": "Sex",  
 "id": "sex\_upon\_outcome",  
 "deletable": False,  
 "selectable": True  
 },  
 {  
 "name": "Age (weeks)",  
 "id": "age\_upon\_outcome\_in\_weeks",  
 "deletable": False,  
 "selectable": True  
 },  
 {  
 "name": "Rec #",  
 "id": "rec\_num",  
 "deletable": False,  
 "selectable": True  
 },  
  
 ],  
 editable=False,  
 filter\_action="native",  
 sort\_action="native",  
 sort\_mode="multi",  
 column\_selectable=False,  
 row\_selectable="single",  
 row\_deletable=False,  
 selected\_rows=[0],  
 page\_action="native",  
 page\_current=0,  
 page\_size=10  
),

This section is lengthy mostly due to renaming columns rather than using the value names as found in the database. For example “Sex” was shortened from the “sex\_upon\_outcome” field. This was done to improve the user experience. The options at the end enable rows to be selected, enable users to use the native table filtering if they so desire, and prevents table fields from being edited.

**Graph layout:**

################ Div line ################  
 html.Hr(), # Div line  
 ################ Charts ################  
  
 # This sets up the dashboard so that your chart and your geolocation chart are side-by-side  
 html.Div(className='row',  
 style={'display': 'flex'},  
 children=[  
  
 html.Div(  
 id='graph-id',  
 className='col s12 m6',  
 style={"width": "50%"}  
 ),  
 html.Div(  
 id='map-id',  
 className='col s12 m6',  
 style={"width": "50%"}  
 )  
 ]),  
  
 ################ Div line ################  
 html.Hr(), # Div line  
 ################ Footer ################  
 html.H1("Rachel Aldava SNHU CS-340 Client/Server Development"),  
])

This section is small because the graphs are defined elsewhere. We assign names to them and specify that they will each take up 50% of the screen so that there will be an even split down the middle.

**Table callback:**

# This updates the table  
@app.callback(  
 Output("datatable-id", "data"),  
 Input("filter", "data")  
)  
def update\_dashboard(my\_filter):  
 records = shelter.read(my\_filter) # Read from the database  
 try:# Try to display the data  
 df = pd.DataFrame.from\_records(records)# Load the data into pandas  
 df.drop(columns=['\_id'], inplace=True) # Use pandas to prune mongo-specific information  
 df.fillna('', inplace=True)  
 except:# If there is an error, display a blank table  
 df = pd.DataFrame.from\_records([])  
  
 return df.to\_dict("records")

Dash components will only change if there is an event called a “callback”. A callback is usually triggered when a value is updated. Callbacks need an input and an output; here we are taking the “filter” data value (from our data store), performing some operations, then outputting something into the data table. In this case, whenever the “filter” variable is updated, this callback will trigger. Here, the filter is the search parameter that we will pass to AnimalCrud.py. When the data is retrieved, it will be converted into a data frame and slightly tweaked before being returned. If there is an error, a blank data set is returned. Once this is returned, the data table will update according to the columns we set up in the layout section.

**Table cell selection callback:**

# This callback will highlight a cell on the data table when the user selects it  
@app.callback(  
 Output('datatable-id', 'style\_data\_conditional'),  
 [Input('datatable-id', 'selected\_columns')]  
)  
def update\_styles(selected\_columns):  
 style\_data = None  
 if selected\_columns != None:  
 style\_data = [{'if': {'column\_id': i}, 'background\_color': '#D2F3FF'} for i in selected\_columns]  
 return style\_data

This callback was bundled with the starter-code; I do not believe I modified it. It simply changes the displayed color of a cell whenever it is clicked. Its inclusion in the project seems… vestigial. I used some ideas from this method to try to modify the colors and styles of the whole page, but I did not produce a color palette which was pleasing to the eye.

**Pie chart update callback:**

@app.callback(  
 Output('graph-id', "children"),  
 [Input('datatable-id', "derived\_virtual\_data")])  
def update\_graphs(viewData):  
 try:  
 # Load the data from the table  
 dff = pd.DataFrame(viewData)  
 breed\_counts\_series = dff['breed'].value\_counts() # count the number of times a breed occurs  
 dff = pd.DataFrame(breed\_counts\_series) # convert to a data frame  
 dff = dff.reset\_index()  
 dff.columns = ['Breed', 'Total']# rename columns  
 # Since the dataframe was already sorted by order of most frequent, we can leverage this by renaming everything'  
 # after a certain index such that less common breeds will be displayed as "other" rather than have  
 # hundreds of different entries on the pie chart  
 if len(dff.index) > 8:  
 dff.loc[8:, 'Breed'] = 'other'  
  
 return html.Div([  
 html.H2('Breed Distribution:', style={'font-size': '18px'}),  
 dcc.Graph(figure=px.pie(dff, values='Total', names='Breed'))  
 ])  
 except:  
 return None

Every time the table is updated, the pie chart is updated. The data is transformed and processed into a more simplified, pie-chart friendly format.

**Map chart update callback:**

# This callback will update the geo-location chart for the selected data entry  
# derived\_virtual\_data will be the set of data available from the datatable in the form of  
# a dictionary.  
# derived\_virtual\_selected\_rows will be the selected row(s) in the table in the form of  
# a list. For this application, we are only permitting single row selection so there is only  
# one value in the list.  
# The iloc method allows for a row, column notation to pull data from the datatable  
@app.callback(  
 Output('map-id', "children"),  
 [Input('datatable-id', "derived\_virtual\_data"),  
 Input('datatable-id', "derived\_virtual\_selected\_rows")])  
def update\_map(viewData, index):  
 try:  
 dff = pd.DataFrame.from\_dict(viewData)  
 # Because we only allow single row selection, the list can  
 # be converted to a row index here  
 if index is None:  
 row = 0  
 else:  
 row = index[0]  
  
 # Austin TX is at [30.75,-97.48]  
 return [  
 dl.Map(style={'width': '1000px', 'height': '500px'},  
 center=[30.75, -97.48], zoom=10, children=[  
 dl.TileLayer(id="base-layer-id"),  
 # Marker with tool tip and popup  
 # Column 13 and 14 define the grid-coordinates for  
 # the map  
 # Column 4 defines the breed for the animal  
 # Column 9 defines the name of the animal  
 dl.Marker(position=[dff.iloc[row, 13], dff.iloc[row, 14]],  
 children=[  
 dl.Tooltip(dff.iloc[row, 4]),  
 dl.Popup([  
 html.H1("Animal Name"),  
 html.P(dff.iloc[row, 9])  
 ])  
 ])  
 ])  
 ]  
 except:  
 return None

Much of this code was supplied by the initial code. I have made small modifications including solving an issue where errors were thrown when no data was supplied, but largely this chart was supplied by the starter code. Whenever the data table is updated, It defines the dimensions of the map, it finds the location data for the animal, then drops a pin at the appropriate coordinate.

**Reset button callback:**

@app.callback(  
 Output("filter\_preset\_radio", "value"),  
 Input('filter\_preset\_reset', "n\_clicks"))  
def preset\_reset(n\_clicks):  
 return None

By far the simplest callback, it simply outputs an input of “None” to the radio value whenever the reset button is pressed.

**Preset filter callback:**

# This sets the custom filter components to preset values based on the value of the filter preset radio  
@app.callback(  
 Output("filter\_species", "value"),  
 Output("filter\_age", "value"),  
 Output("filter\_outcome", "value"),  
 Output("filter\_age\_options", "value"),  
 Output("filter\_sex", "value"),  
 Output("filter\_breed", "value"),  
 Input('filter\_preset\_radio', "value"))  
def preset\_apply(preset\_value):  
 # If there is no value, then display every animal  
 if preset\_value == None:  
 filter\_species = []  
 filter\_breed = []  
 filter\_outcome = []  
 filter\_age\_options = []  
 filter\_sex = []  
 filter\_age = [0, 1043]  
   
 # If water rescue is selected, output the relevant values to the custom filter components.  
 elif "Water Rescue" in preset\_value:  
 filter\_species = ["Dog"]  
 filter\_breed = ["Labrador Retriever",  
 "Labrador Retriever Mix",  
 "Chesa Bay Retr Mix",  
 "Labrador Retriever/Chesa Bay Retr",  
 "Newfoundland Mix",  
 "Newfoundland/Labrador Retriever",  
 "Labrador Retriever/Newfoundland"]  
 filter\_outcome = []  
 filter\_age\_options = ['Exclude deceased', 'Use "age at outcome" instead of current age']  
 filter\_sex = ["Intact Female"]  
 filter\_age = [26, 156]  
  
 # If mountain rescue is selected, output the relevant values to the custom filter components.  
 elif "Mountain Rescue" in preset\_value:  
 filter\_species = ["Dog"]  
 filter\_breed = ["German Shepherd",  
 "German Shepherd Mix",  
 "Alaskan Malamute",  
 "Alaskan Malamute Mix",  
 "German Shepherd/Alaskan Malamute",  
 "Old English Sheepdog",  
 "Siberian Husky",  
 "Siberian Husky Mix",  
 "German Shepherd/Siberian Husky",  
 "Siberian Husky/German Shepherd",  
 "Rottweiler",  
 "Rottweiler/Siberian Husky",  
 "Rottweiler Mix",  
 "Rottweiler/German Shepherd",  
 "German Shepherd/Rottweiler"]  
 filter\_outcome = []  
 filter\_age\_options = ['Exclude deceased', 'Use "age at outcome" instead of current age']  
 filter\_sex = ["Intact Male"]  
 filter\_age = [26, 156]  
  
 # If mountain rescue is selected, output the relevant values to the custom filter components.  
 elif "Disaster Rescue" in preset\_value:  
 filter\_species = ["Dog"]  
 filter\_breed = ["German Shepherd",  
 "German Shepherd Mix",  
 "Rottweiler",  
 "Rottweiler Mix",  
 "Rottweiler/German Shepherd",  
 "German Shepherd/Rottweiler",  
 "Doberman Pinsch",  
 "Doberman Pinsch Mix",  
 "German Shepherd/Doberman Pinsch",  
 "Bloodhound",  
 "Bloodhound Mix",  
 "Golden Retriever",  
 "Golden Retriever Mix"]  
 filter\_outcome = []  
 filter\_age\_options = ['Exclude deceased', 'Use "age at outcome" instead of current age']  
 filter\_sex = ["Intact Male"]  
 filter\_age = [20, 300]  
  
 return filter\_species, filter\_age, filter\_outcome, filter\_age\_options, filter\_sex, filter\_breed

This callback is triggered whenever the preset radio component is updated. It will define values for the search parameter components, then set them. These different values were determined based off of the client’s specifications.

**Custom Filter Parsing Callback:**

# This callback processes the filter components input and converts it into a valid query for PyMongo  
@app.callback(  
 Output("filter", "data"),  
 Input("filter\_species", "value"),  
 Input("filter\_breed", "value"),  
 Input("filter\_outcome", "value"),  
 Input("filter\_age", "value"),  
 Input("filter\_age\_options", "value"),  
 Input("filter\_sex", "value"),  
)  
def update\_filter(filter\_species, filter\_breed, filter\_outcome, filter\_age, filter\_age\_options, filter\_sex):  
 # Example query strategy:  
 #  
 # species\_other = {"animal\_type": "Other"}  
 # species\_bird = {"animal\_type": "Bird"}  
 # outcome\_euthanasia = {"outcome\_type" : "Euthanasia"}  
 # outcome\_adoption = {"outcome\_type": "Adoption"}  
 # query\_species = {"$or": [species\_other, species\_bird]}  
 # query\_outcome = {"$or": [outcome\_euthanasia, outcome\_adoption]}  
 # query\_full = {"$and": [query\_species, query\_outcome]}  
 # for doc in shelter.read(query\_full):  
 # print(f"\n{doc}\n")  
 #  
 # In the above example, the query was pieced together by stacking dictionaries inside of "$or" keyed dictionaries,  
 # then keying each of those or dictionaries inside an "$and" dictionary.  
 # PyMongo will read this as "any species of (other or bird) that has an outcome of (euthanasia or adoption)  
 #  
 #  
 # We will now begin:  
 filter\_and = []# An empty "and" dictionary which will later add "or" dictionaries to  
  
 # Apply species filter "or" dictionary  
 if filter\_species is not None and len(filter\_species) > 0:  
 filter\_and.append({"$or": [{"animal\_type": species} for species in filter\_species]})  
  
 # Apply breed filter "or" dictionary  
 if filter\_breed is not None and len(filter\_breed) > 0:  
 filter\_and.append({"$or": [{"breed": breed} for breed in filter\_breed]})  
  
 # Apply outcome filter "or" dictionary  
 if filter\_outcome is not None and len(filter\_outcome) > 0:  
 filter\_and.append({"$or": [{"outcome\_type": outcome} for outcome in filter\_outcome]})  
  
 # Apply sex filter "or" dictionary  
 if filter\_sex is not None and len(filter\_sex) > 0:  
 filter\_and.append({"$or": [{"sex\_upon\_outcome": sex} for sex in filter\_sex]})  
  
 # Apply deceased filter (not an or filter; there are only two possible states):  
 if 'Exclude deceased' in filter\_age\_options:  
 filter\_and.append({"is\_deceased": False})  
  
 # Apply age filters  
 if filter\_age is not None and len(filter\_age) > 0:  
 # Default: we use age upon outcome, in units of weeks  
 age\_column = 'age\_upon\_outcome\_in\_weeks'  
 age\_min = filter\_age[0]  
 age\_max = filter\_age[1]  
  
 # If option is unchecked, we use age based on date-of-birth, which is measured in years  
 if 'Use "age at outcome" instead of current age' not in filter\_age\_options:  
 age\_column = 'age'  
 weeks\_per\_year = 52.1429  
 age\_min /= weeks\_per\_year  
 age\_max /= weeks\_per\_year  
  
 filter\_and.append({age\_column: {"$gt": age\_min}})# for PyMongo, "$gt" means greater than  
 # If there is a maximum age  
 if filter\_age[1] < 1040:  
 filter\_and.append({age\_column: {"$lt": age\_max}})# for PyMongo, "$lt" means lesser than  
  
 # By default, our query will be blank  
 query = {}  
 # If the "and" list has any elements, we can safely add an "and" component to the search query  
 if len(filter\_and) != 0:  
 query = {"$and": filter\_and}  
  
 return query

This is the primary sauce of the project; It’s the big chalupa. It parses the filter component values as an input, translates it into a query that PyMongo can work with, and stores it in the “filter” store component that we previously defined. Remember that the table is updated whenever the filter store is updated.

PyMongo queries are dictionaries. We can use the component values to build a query. It might be a bit hard to wrap your head around, but the code is simple.

A simple query looks like this: {"animal\_type": "Dog"}

An OR query is like this: {"$or": [query1, query2]}

So, a compound query which shows either dogs or cats would be:

{"$or": [{"animal\_type": "Dog"}, {"animal\_type": "Cat"}]}

Which would give us the portion of the filter for species. The species\_filter value might look like [“Dog”, “Cat”], so we would need code which could parse that list into the dictionary above. This is easily done with basic python skills, including the method I used above.

An AND query is like this: {"$and": [query1, query2]}. So, in order to combine our filters together, we need to stack them up into a list, then simply create that dictionary with that key.

A PyMongo query {"animal\_type": "Dog", "animal\_type": "Cat"} is also an AND search, but building an “$and” query seemed to have more explicit logic.

## Roadmap/Features/Challenges:

## As mentioned previously, I wanted to add a “dark mode” theme to the web page, but I couldn’t find a color palette which pleased the eye. Part of the issue is that the red in the company logo is not a very good color for text since it resulted in low contrast with any given shade of black, gray, or white. According to basic color theory, a blue and slightly green background color such as seafoam or Caribbean blue would produce the highest contrast, but such a background color is out of style. Given a full day, my limited artistic skills could probably arrive at a new theme but for now the default is sufficient. Dash does supply its own “darkly” theme, but this would require installing additional dependencies.

* In a previous version of this project, I filtered out “breed” options in the dropdown menu based off of the current selection in the “species” dropdown such that if a user selected “cat” without selecting “dog”, they would not have the option to select a “Doberman” “Cat”. The reason this feature was removed is that it caused an issue whenever the preset filters were applied. Whenever the species and breed components were updated by the preset callback, the breed option pruning was triggered and the breed selection was reset. I spent some time researching the issue and thought of several strategies, but I placed a low priority on implementing them and have since run out of development time.
  + Strategy 0: Do nothing, try to implement it again, and see if my modifications to other sections of the code resolved the issue
  + Strategy 1: Investigate the proposed solution <https://community.plotly.com/t/dynamically-update-dropdown-options-menu-list-without-clearing-previously-selected-options/79452/4>
  + Strategy 2: Change the species dropdown menu to a radio item component, Hard-code breed selection options with one dropdown menu for each species. Dynamically hide or show these species-specific breed dropdown menus based on whether or not the user selected the species in the radio item component. If there are no options to update, the issue of value deselection whenever options update is eliminated.
  + Strategy 3: Allow users to type any value in the breed section.

Currently strategy 2 is my favorite. If a user is looking through breeds of “cats” and “dogs” together, the breed option “domestic shorthair” is not clearly associated with either species; it’s not clear that it is a cat-specific breed.

* The data within the ACC database is not clean. The worst example is in the “breed” value, which does not contain any standardization. The breeds "German Shepherd", "German Shepherd Mix", "Siberian Husky", "Siberian Husky Mix", "German Shepherd/Siberian Husky", "Siberian Husky/German Shepherd" all appear in the database, whenever these breeds could be cleaned up into something better. For example, {“breed”: [“German Shepherd”, “Siberian Husky”] and “is\_mixed\_breed” : False} would allow for my program to work with and parse animals in a much more robust manner.
  + To that end, a hardened creation and update method could ensure that new entries remain in workable format
* The speed of queries could be improved with indexes. However, this is a database design decision which would need to be made based off of use-case. Are animals added to the database more frequently than the database is searched? If so, spending compute time at document insertion seems like a bad choice. If not, saving compute time with the web page queries seems like a good improvement.

## Contact

Rachel Aldava

https://github.com/RachelAldava

rachel@aldava.net