# Project Two README

# Grazioso Salvare/AAC Dashboard Application

## About the Project/Project Title

This project is a full web-based application built with a python “glue” layer to facilitate interaction between an interface and a MongoDB database.

## Motivation

While interacting through command line is effective for MongoDB databases, most users do not have the familiarity and experience to do so effectively. Therefore, in order to have a database that its userbase can access properly, an interface is needed. Functionality was also implemented to aid the shareholder in accomplishing their mission, namely preset filter options and visualizations to simplify data consumption.

## Getting Started

To start locally, you first need to update the MongoClient initialization information for the desired database. To do so, open AnimalShelter.py and make the necessary changes to:

* USER
* PASS
* HOST
* PORT
* DB
* COL

From there, simply include it in your python project, create a new AnimalShelter object, and call functions as needed. The AnimalShelter class is designed to construct a new object with a simple pass of a username and password as parameters. The other settings are hardcoded.

## Installation

This project required the following tools:

* IDE
  + For development of the .py file. I used Spyder in this instance. IDEs allow for easier and faster development, as they are purpose built for code development.
* Jupyter Notebooks
  + For testing and communication purposes. As a rapid testing tool it does well in building tests through the python class file in order to test the proper functioning of the application. It also allows for easy sharing with others on how development progressed and the testing done.
* Pymongo
  + This library is used to interact with MongoDB databases via python. It is necessary for the two languages to interact properly.
* MongoDB
  + This is the base database system layer. MongoDB is built for rapid sorting and searching of databases. It has prebuilt functions for each element of the CRUD process. For instance, the find() method is instrumental in the filtering that this application does in the read process.
* Panda
  + Panda is a python library that is built for data management. While the database itself is created and stored—and ultimately manipulated—in MongoDB, the use of panda dataframes and the built-in functions it has are instrumental in utilizing python as the glue layer like we do. In particular, panda allows us to make rapid, filtered dataframes based on search criteria that we can then use to analyze the data and send it to graphs and visuals.
* Plotly
  + Plotly is another library that we utilize for our graphs. It is a powerful tool built for data visualization, and while we only used its pie graph functionality, there are many more built-in graph type we could utilize.
* Dash
  + Dash is the library we utilize for the user interface. It allows us to build a webpage with prebuilt widgets—such as buttons, dropdowns, datatables, and more—and create events that can run python code from our glue layer. Each layer is vital, but Dash is essential for giving the application the desired functionality and useability.
* Dash Leaflet
  + This tool allows us to create the maps for the geolocation portion of this project.

**AnimalShelter.py**

## Usage

MongoDB was chosen as a standard database management tool. It has built in tools to sort and search through data. Of particular note, we utilize the insert\_one(), find(), update\_many(), and delete\_many() functions.

Pymongo is needed to serve as a middleman between the mongdb database and the client layer.

### Code Example

Initialization (variables must be updated)

def \_\_init\_\_(self):

        #

        # Connection Variables

        #

        USER = 'aacuser'

        PASS = 'userpass'

        HOST = 'nv-desktop-services.apporto.com'

        PORT = 31057

        DB = 'AAC'

        COL = 'animals'

        #

        # Initialize Connection

        #

        self.client = MongoClient('mongodb://%s:%s@%s:%d' % (USER,PASS,HOST,PORT))

        self.database = self.client['%s' % (DB)]

        self.collection = self.database['%s' % (COL)]

Create() Function

Here we utilize the insert\_one() function.

def create(self, data):

        if data:

            self.database.animals.insert\_one(data)  # data should be dictionary

            return True

        else:

            raise Exception("Nothing to save, because data parameter is empty")

            return False

Read() Function

Here we utilize the find() function.

def read(self, data):

        if data:

            result = self.database.animals.find(data)

            return list(result)

        else:

            result = self.database.animals.find({})

            return list()

Update() Function

Here we utilize the update\_many() function.

# Create method to implement the U in CRUD

    def update(self, data, updateData):

        if data is not None:

            result = self.database.animals.update\_many(data, {"$set": updateData})

            return result

        else:

            return set()

Delete() Function

Here we utilize the delete\_many() function.

# Create method to implement the D in CRUD

    def delete(self, data):

        if data is not None:

            result = self.database.animals.delete\_many(data)

            return result

        else:

            return set()

### Tests

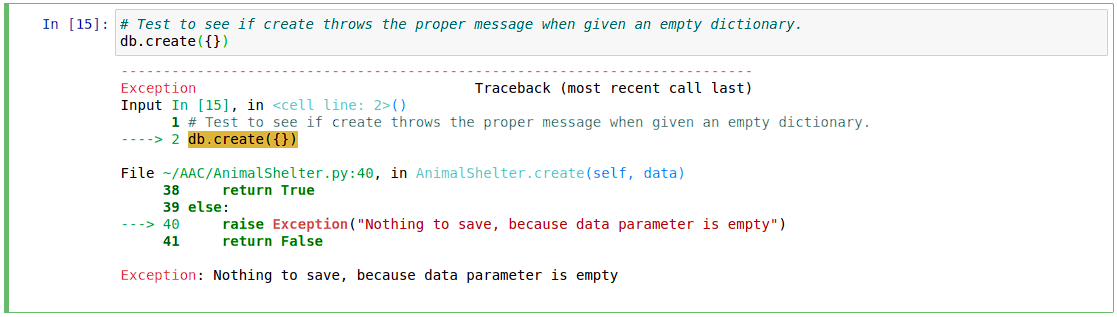
We need to test six things:

1. Creation with proper input:
   1. db.create(data)
   2. where db is the AnimalShelter object and data is a new document to enter.
2. Creation with empty dictionary:
   1. db.create({})
   2. where db is the AnimalShelter object
3. Read with a key/value pair
   1. db.read({key: value})
   2. where db is the AnimalShelter object
4. Read with empty dictionary
   1. db.read({})
   2. where db is the AnimalShelter object
5. Update Functionality:
   1. db.update(data, updateData)
   2. where db is the AnimalShelter object
6. Delete Functionality:
   1. db.delete(data)
   2. where db is the AnimalShelter object

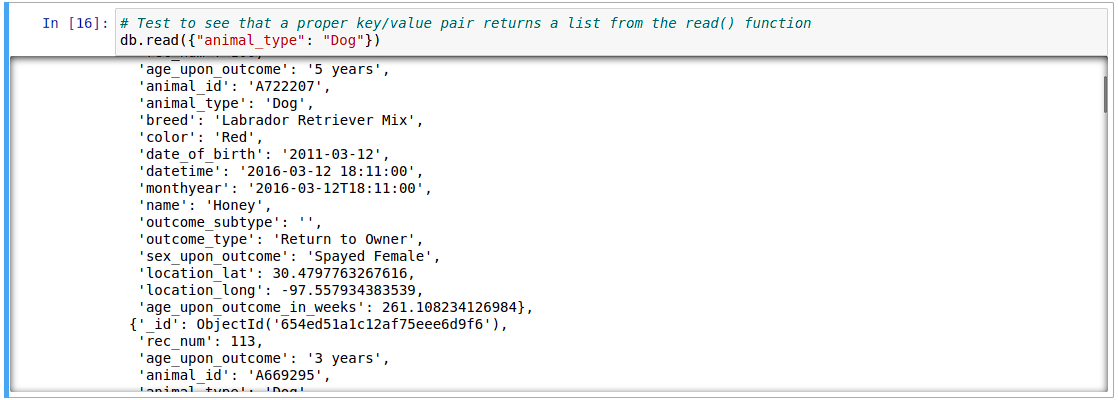
### Screenshots

create() function with True return

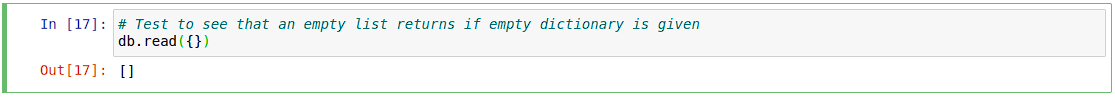
create() function with empty parameter



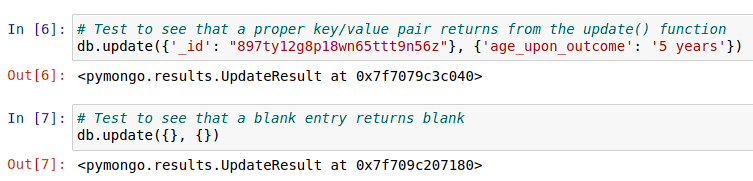
read() function with list return



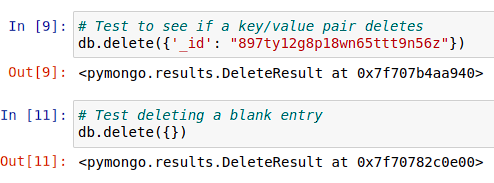
read() function with empty parameter and empty list return



update() function tests



delete() function tests



**Dash Application**

These imports are needed for the application to function:

# 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

from dash import ctx

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

The application relies on an AnimalShelter object for it use our python CRUD layer:

# change animal\_shelter and AnimalShelter to match your CRUD Python module file name and class name

from AnimalShelter import AnimalShelter

###########################

# Data Manipulation / Model

###########################

# FIX ME update with your username and password and CRUD Python module name

username = "aacuser"

password = "userpass"

# Connect to database via CRUD Module

db = AnimalShelter(username, password)

# class read method must support return of list object and accept projection json input

# sending the read method an empty document requests all documents be returned

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

Notice this line: It is important to keep Dash running happily.

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

Later, when we reload the database after working on it, we need to expand that statement:

# loops through the columns in df. If \_id exists, deletes it.

    # if \_id is deleted without this check like in the starting code, and error occurs

    for col in df:

        if col == '\_id':

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

    return df.to\_dict('records')

This code ensures that there is an ‘\_id’ column to delete before it tries. This avoids an error that was previously present in the application.

This is just a snippet of the html creation through Dash:

app.layout = html.Div([

    html.Div(id='header-div', style={'display': 'flex'},

             children=[html.A([html.Img(

                        src='data:image/png;base64,{}'.format(encoded\_image.decode()),

                        alt='Grazioso Salvare Logo',

                        height='150px',

                        width='150px',

                        )], href='www.snhu.edu'),

                       html.Center(html.B(html.H1('AAC/Grazioso Salvare Dash - Nick Nevins'))),]),

    html.Hr(),

    html.Div(

#FIXME Add in code for the interactive filtering options. For example, Radio buttons, drop down, checkboxes, etc.

        [

            html.Button('Water', id='water\_rescue\_button', n\_clicks=0),

            html.Button('Mountain or Wilderness', id='mountain\_wilderness\_button', n\_clicks=0),

            html.Button('Disaster or Individual Tracking', id='disaster\_tracking\_button', n\_clicks=0),

            html.Button('Reset Search', id='reset\_button', n\_clicks=0)

    ]),

This (and more) gives us the interface shown later.

Dash utilizes callbacks and functions in order to execute code based on user input. A callback looks like this:

@app.callback(Output('datatable-id','data'),

              [Input('water\_rescue\_button', 'n\_clicks'),

               Input('mountain\_wilderness\_button', 'n\_clicks'),

               Input('disaster\_tracking\_button', 'n\_clicks'),

               Input('reset\_button', 'n\_clicks'),])

While the function looks like so:

def update\_dashboard(water\_rescue\_button, mountain\_wilderness\_button, disaster\_tracking\_button, reset\_button):

## FIX ME Add code to filter interactive data table with MongoDB queries

    queryHolder = {}       # an empty dicitonary for the read() function

    breedHolder = []       # an empty array to hold the requested breeds

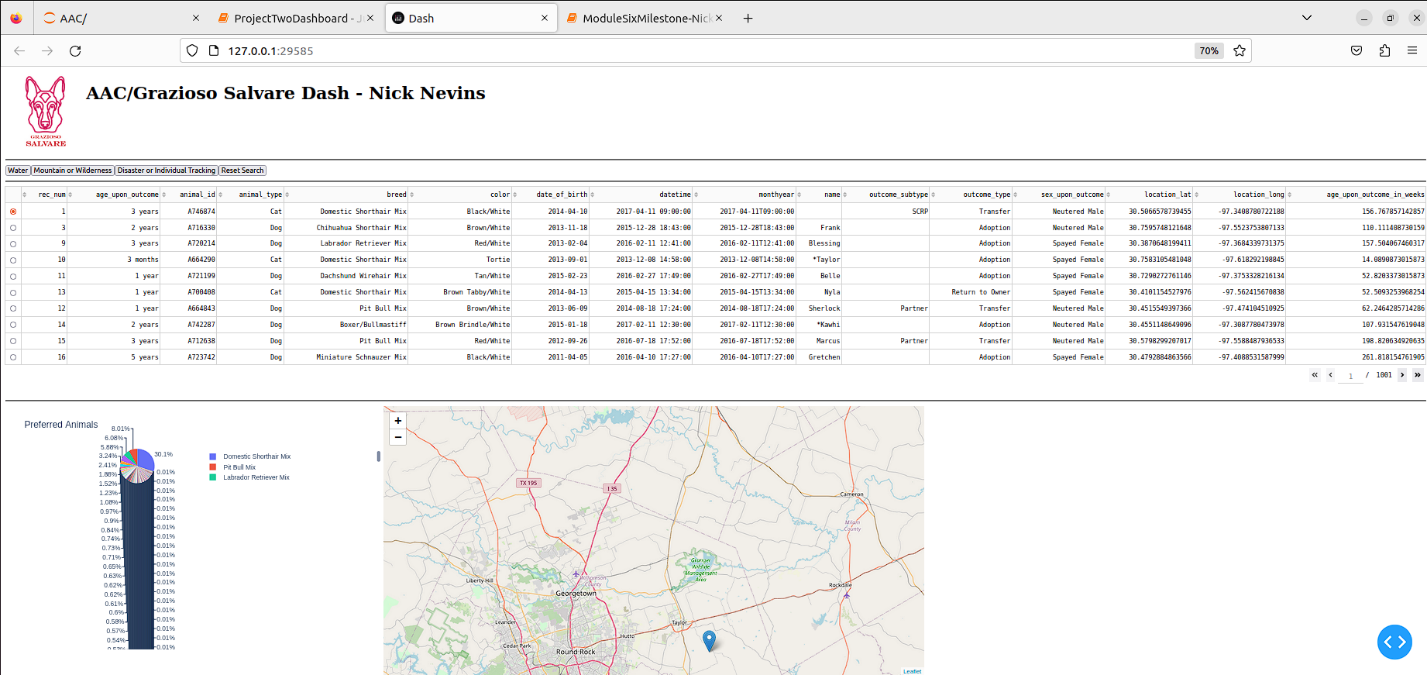
    sexHolder = ''         # an empty string to hold the preferred sex

    ageHolder = [0.0, 0.0] # an empty array for the age min and max

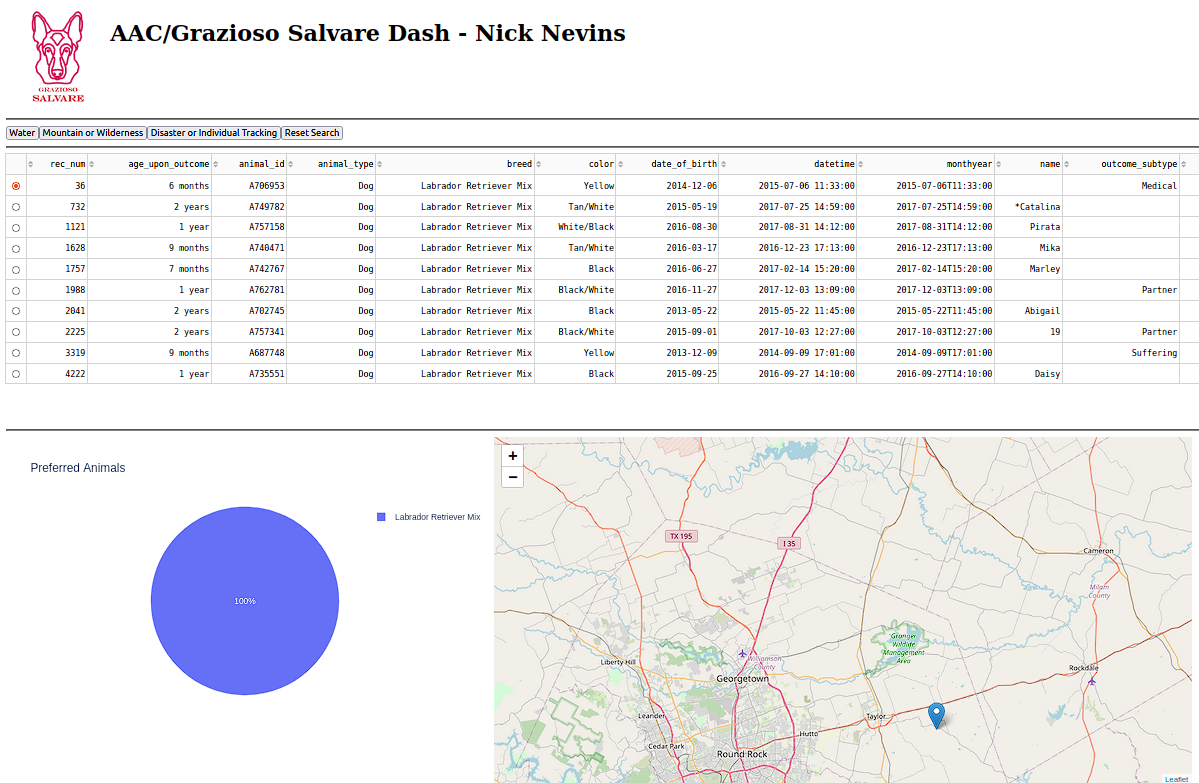
Function shortened for concision. Update\_dashboard is one of the main functions. It holds the controls for the buttons that filter the results by rescue type. It uses 4 members, queryHolder, breedHolder, sexHolder, and ageHolder. When a button is pressed, the function detects which button was pressed utilizing the dash.ctx library’s .triggered\_id() function. Then it provides the appropriate filters to each of the 3 Holders. Those three are then combined into queryHolder, which then is passed into the read() function from the AnimalShelter.py class. It first checks for initial load or the reset button press, and passes through queryHolder as a blank dictionary instead if true. This loads the full database.

**Screenshots**

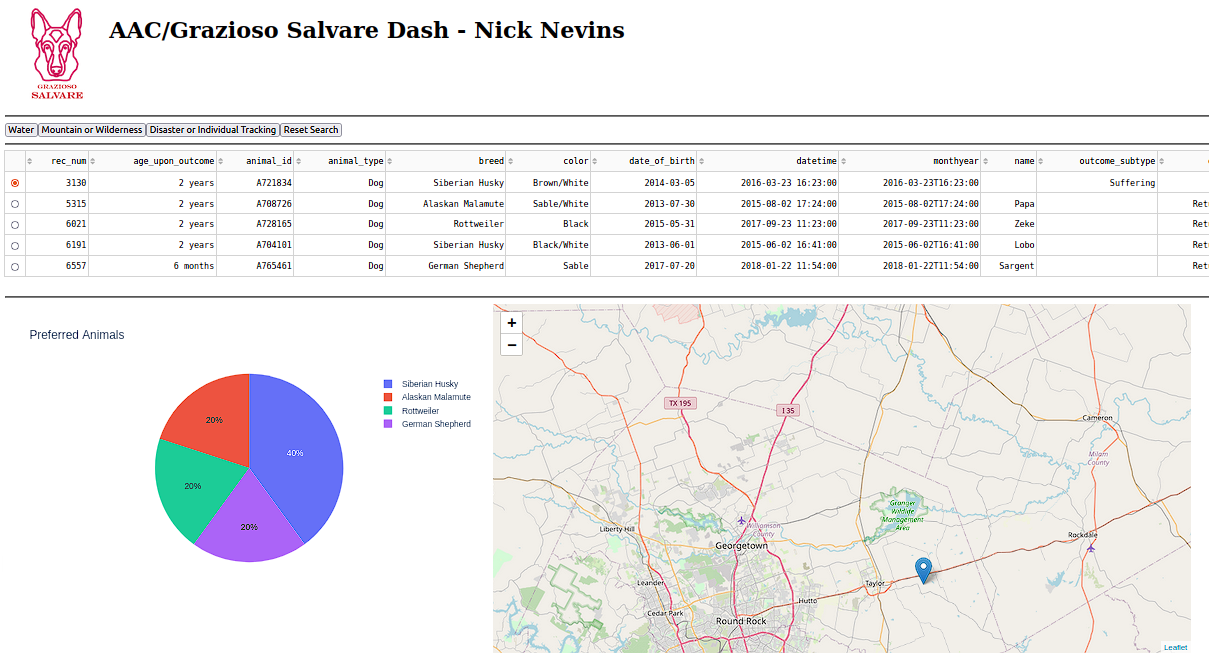
The application’s initial load state:



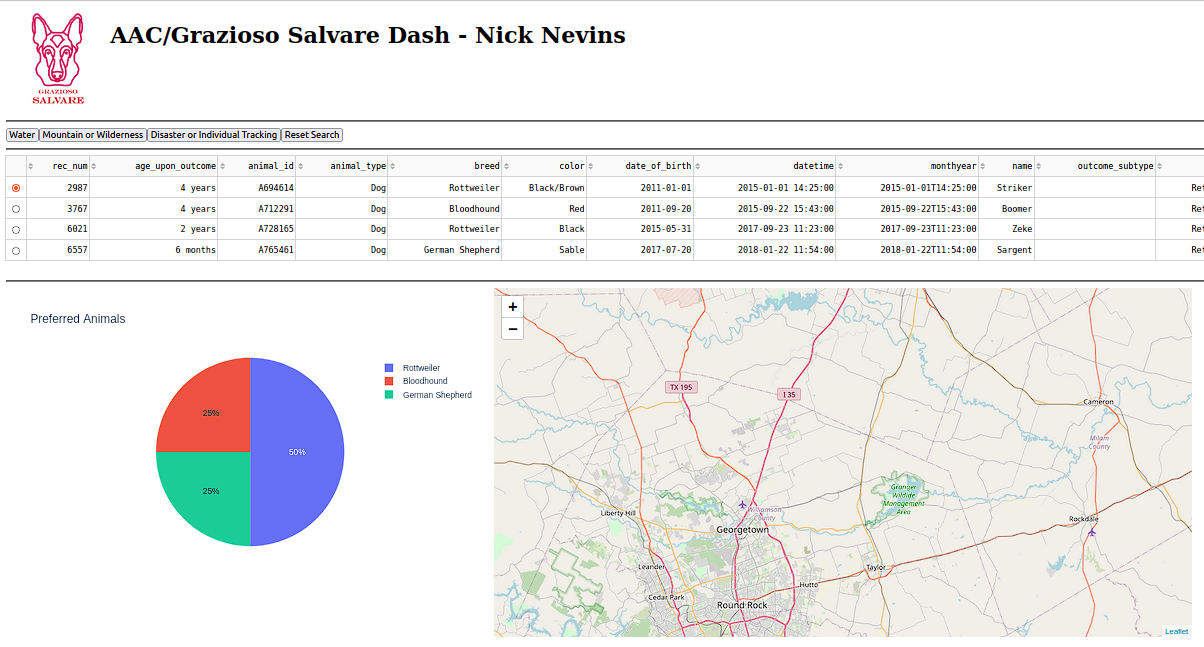
The water rescue filter button pressed:



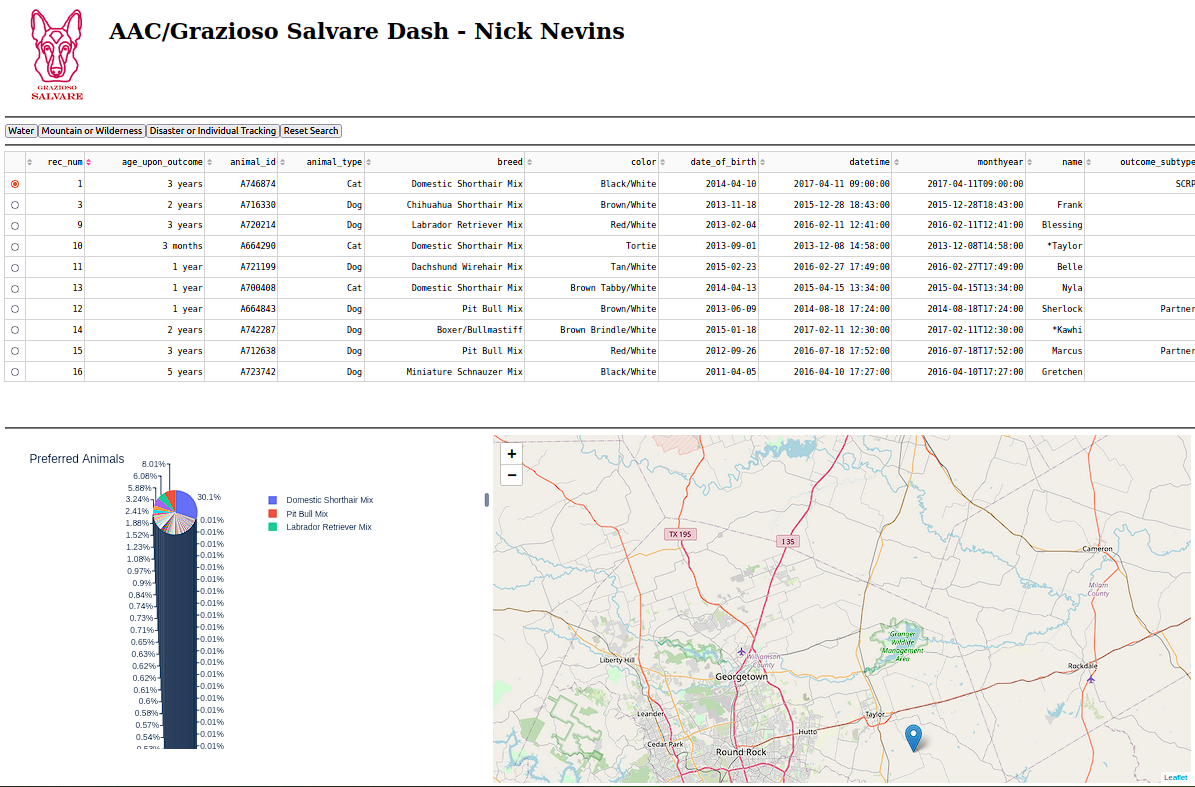
The mountain and wilderness rescue filter button pressed:



The disaster and individual tracking filter button pressed:



The reset button pressed:



**Roadmap/Features (Optional)**

While no further development is currently planned, we could look to add the following functionality:

* User sign in and validation through the UI
* UI features to set up or change the database
* Multiple database support, say for different cities outside the greater Austin area

**Development Path:**

*Approach:*

Development can be broken into three main stages.

* Database development
  + MongoDB was set up and the database was built.
  + CSV imported and collections and documents created
  + Authentication system set in place
* Python CRUD Layer Development
  + Pymongo was utilized to connect with the database
  + Constructor authentication created
  + Create method created utilizing the insert\_one MongoDB function
  + Read method created utilizing the find() MongoDB function
  + Update method created utilizing the update\_many() MongoDB function
  + Delete method created utilizing the delete\_many() MongoDB function
* Dash Interface Development
  + Authentication set up and verified
  + Data table created from the AnimalShelter database
  + Header created
  + Sort buttons added and logic implemented
  + Geolocation map added
  + Pie graph added

*Roadblocks*

There were certain stages that caused some trouble, but creative thinking got me around those.

1. Automatic ‘\_id’ column removal, while necessary, caused errors as initially implemented. Filtered datatables would not have an \_id column, and that caused the error. Therefore, I decided that I needed to loop through the column names and check for an \_id column. If found, it would be removed. This fixed the error.
2. I had a major issue where I was not getting results from the read() function if no input was given, which meant I couldn’t get a full database loaded into the datatable. After much fruitless effort, I realized that I was returning a blank set instead of the results of a find(all) query.
3. I had a bug where the delete() function inadvertently deleted the entire database. I corrected that failed logic.
4. I had issues with the compound search statements for the read() function. This was caused by unfamiliarity with the tools, and was rectified with research and experimentation.
5. Similarly, I had issues with the implementation and updating of the pie graph. Research and experimentation aided in rectifying that.
6. I had some issues with the datatable loading, then clearing itself. After a lot of thought and consideration, I realized it had to do with the logic I built for detecting button presses. Inserting a check for initial load at the beginning solved the issue.

## Contact

Your name: Nicholas Nevins