# CS 340 README for Grazioso Salvare’s Custom Dashboard

# About the Project

The goal of this project was to provide Grazioso Salvare with a useful, web-based dashboard that would enable them assess animals in the Austin Animal Shelter that meet their goals as a business.

This project consists of three layers:

* A MongoDB layer which contains the data needed for visualization
* A Python middle-ware layer that is used for control
* A Plotly-, Dash-, Leaflet-based visualization layer to produce the html

Using the Pymongo driver, the Python middleware enables Create-Read-Update-Delete (CRUD) operations in MongoDB through the use of a custom class. All of the abstracted getter/setter methods required for CRUD operations and MongoDB startup will be included in the class.

**Component Selection Methodology**

MongoDB was selected over SQL because of its versatility and ease of use. Because MongoDB employs a dynamic schema, errors that could arise during an update or create operation are prevented. For example, if an age with decimals is entered, it won't be typed as an integer. Furthermore, a web environment is a better fit for the JSON format from MongoDB.

While MongoDB does not promise the same level of ACID qualities as SQL, this issue is disregarded as this tool is not expected to have many transactions.

Python was selected because it offers a wide range of capabilities for integrating the front end and back end. For instance, the middle-ware layer and CRUD class are utilized by the Pymongo driver to communicate with MongoDB. Using the Plotly-Dash and Dash Leaflet tools, Python can also create dashboards and other widgets with an HTML/JavaScript foundation.

The Dash Leaflet and Plotly-Dash packages are wrappers for JavaScript-based tools that communicate with Python via their corresponding APIs. These libraries enable the generation of the raw HTML code and CSS required to create the data table and widgets, as well as the SVG (pie chart) and tiled-PNG (map) formats.

**CRUD Class**

While utilizing any database, CRUD operations are commonplace, preparing these queries and managing errors can be time-consuming. These actions have been abstracted to straightforward get and set methods to help reduce the tediousness of preparing each new query or re-coding for each error category.

The class contains the following methods:

• Constructor()

• create(data)

• read(criteria)

• update(query, new\_values)

• delete(query)

For more information on the CRUD class refer to the specific readme.

# Installation/Pre-Conditions

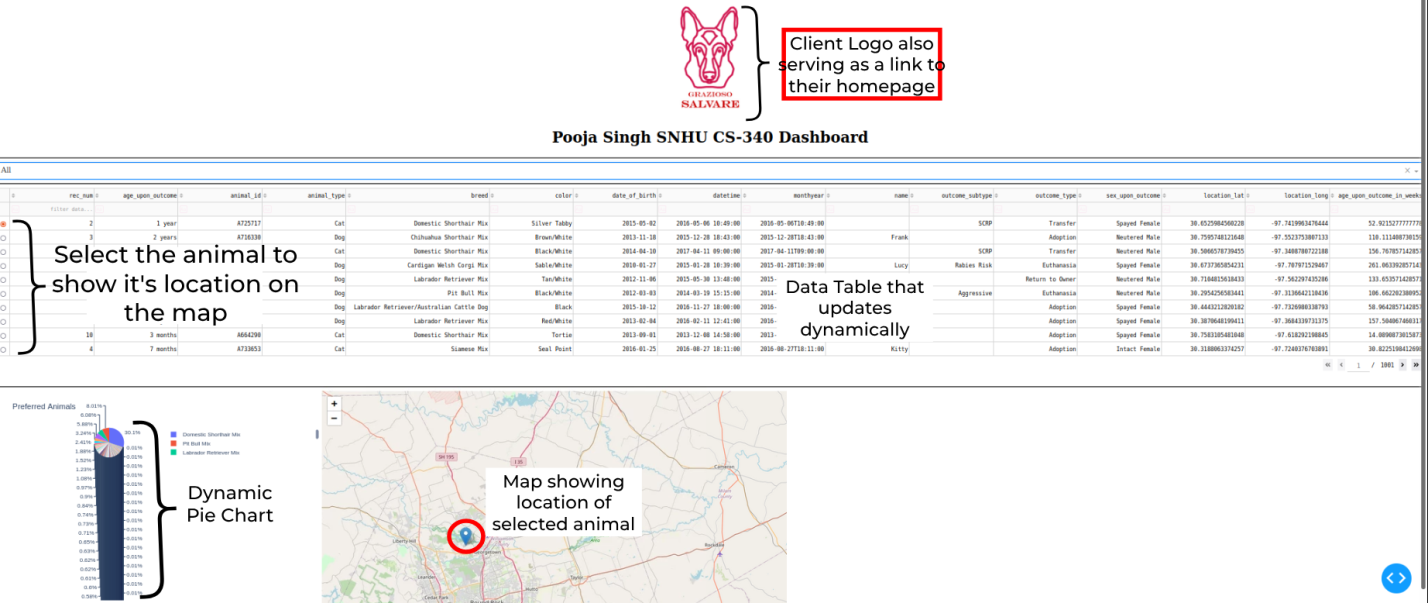
To use this software the following tools must be installed:

* Python: [link](https://www.python.org/downloads/)
* Pymongo: [link](https://pymongo.readthedocs.io/en/stable/)
* MongoDB: [link](https://www.mongodb.com/)
* Plotly Dash: [link](https://dash.plotly.com/)
* Dash Leaflet: [link](https://dash-leaflet.herokuapp.com/)
* Data available in a Mongo database collection, see example on importing a CSV
* User account with read/write permissions

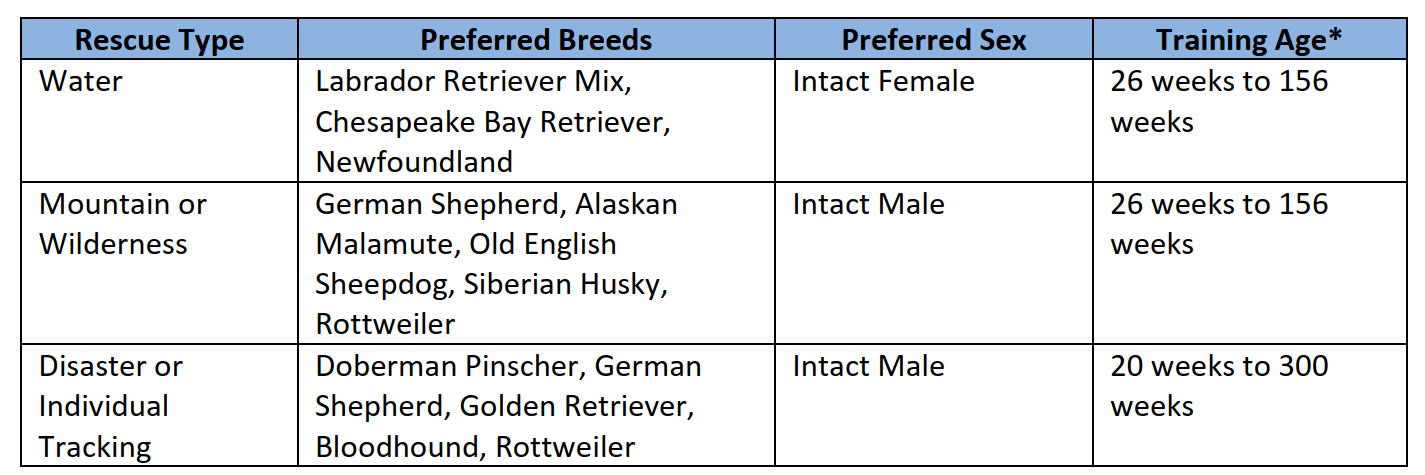
**Product Walkthrough**

Grazioso Salvare requested the dashboard to have a few key components:

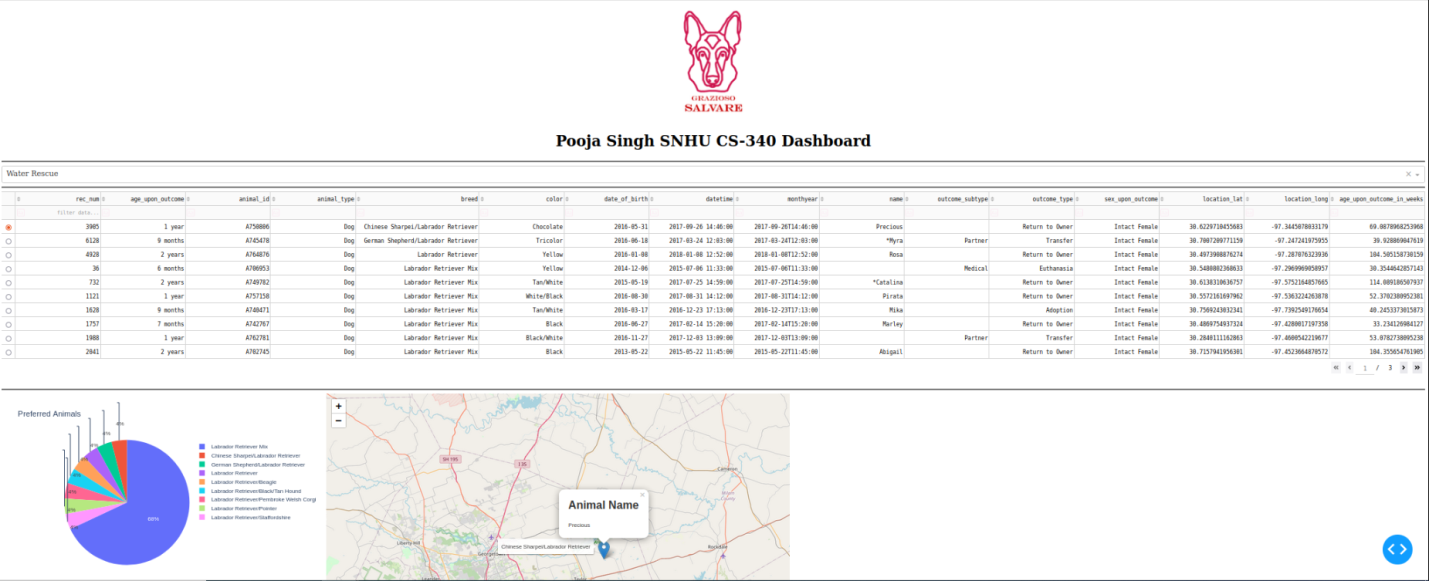
* Branding with logo
* A link to their website
* A data table, with custom filters, containing the information about the animals at the Austin Animal Shelter
  + Custom filters provide targeted animals for specific business needs, including:
    - Water Rescue
    - Mountain or Wilderness Rescue
    - Disaster Rescue or Individual Tracking
* A pie chart showing the breakdown of available breeds
* A map with the location of a selected animal

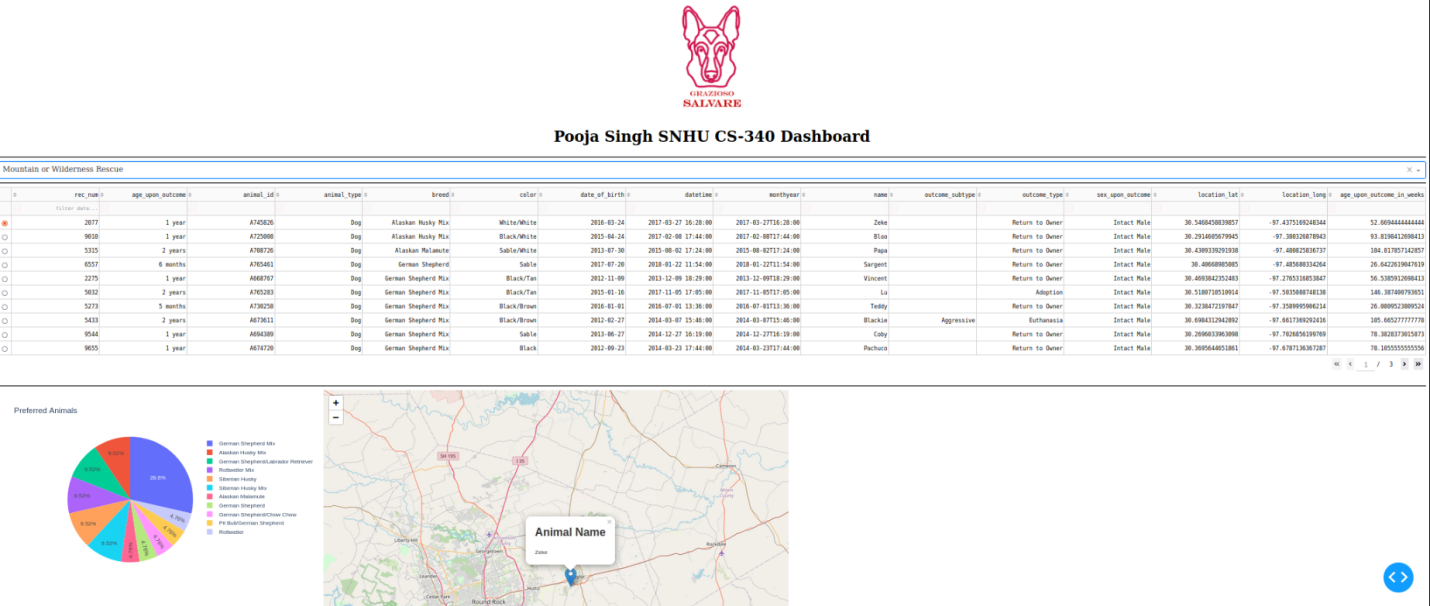


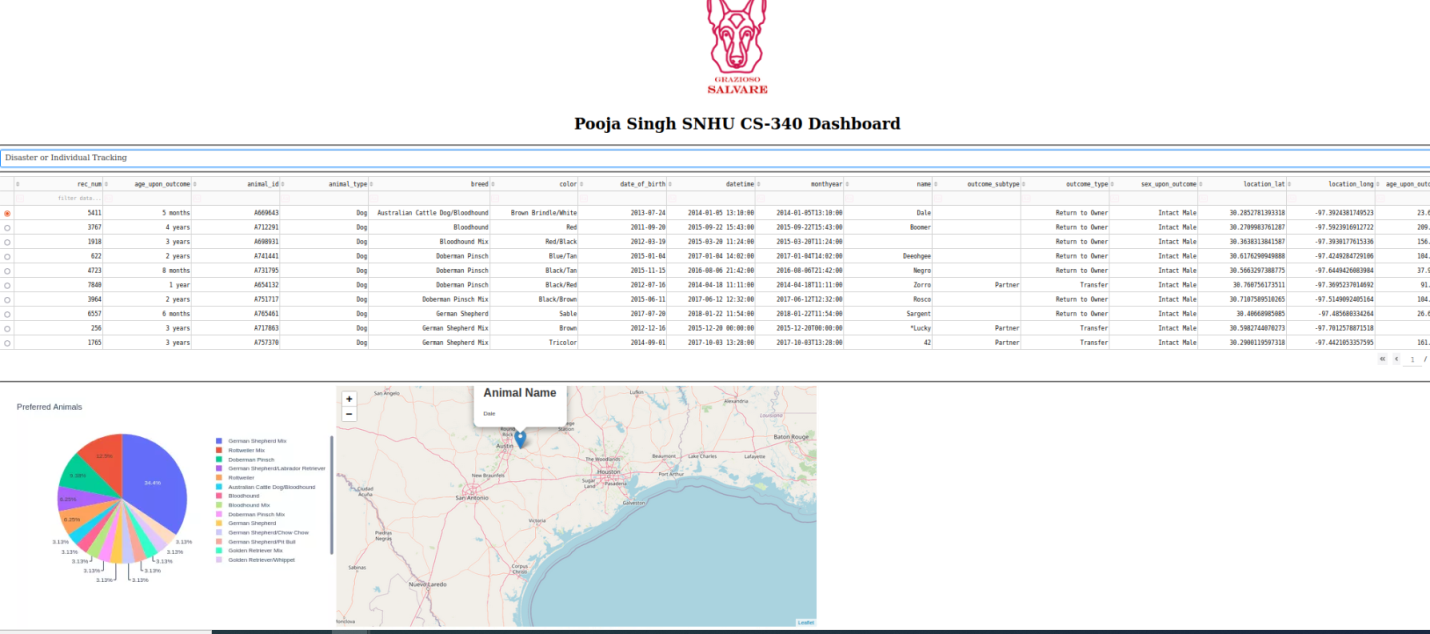
The custom filters are set with the specifications provided by the client, shown below:



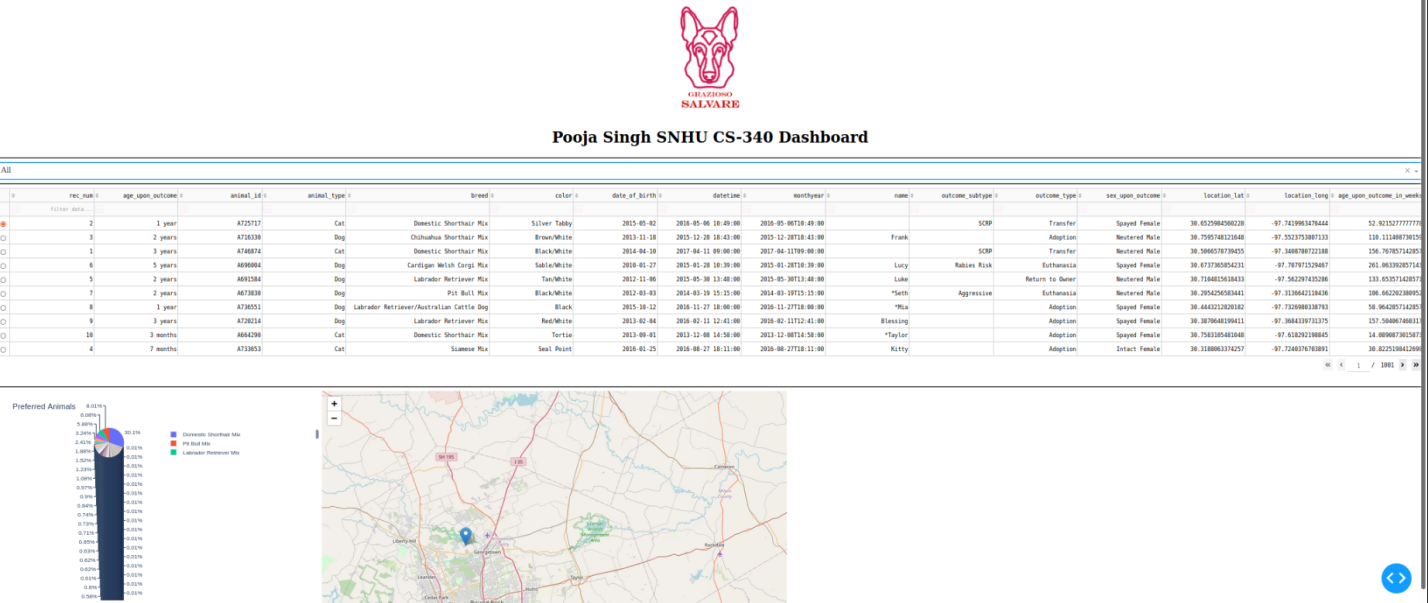
If a user chooses a new filter the pie chart and data table are updated:



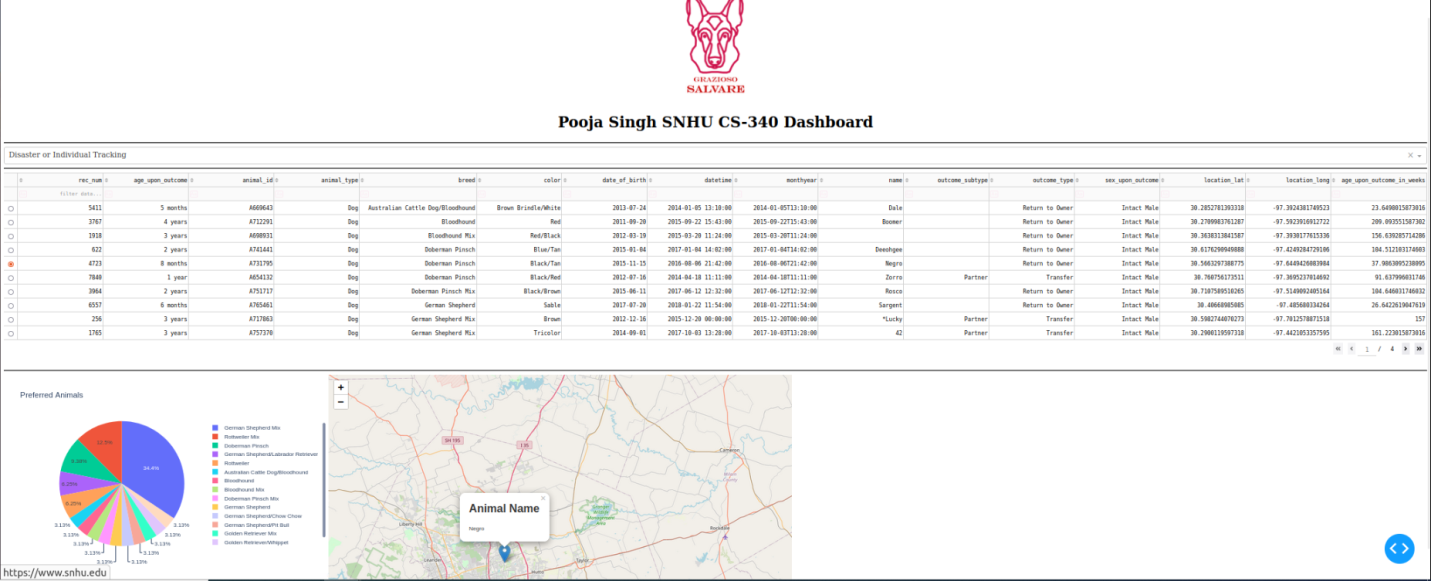




All the widgets are returned to their original, unfiltered state when “All” is selected in the filter options:



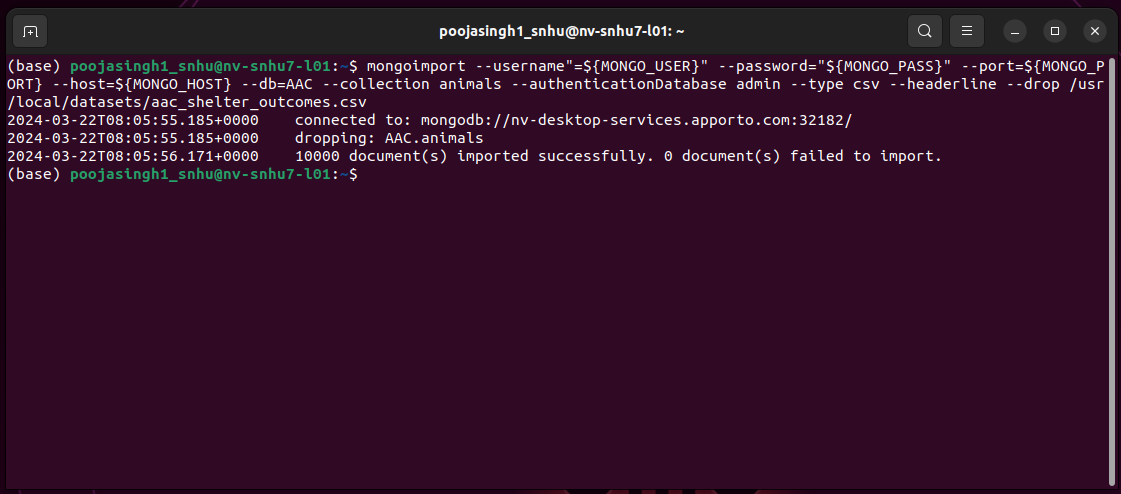
If a particular animal is selected its location is shown on the map:



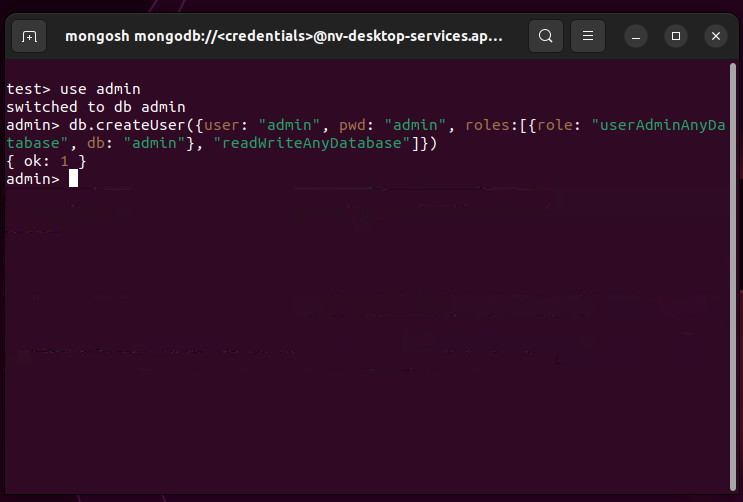
# Project Recreation

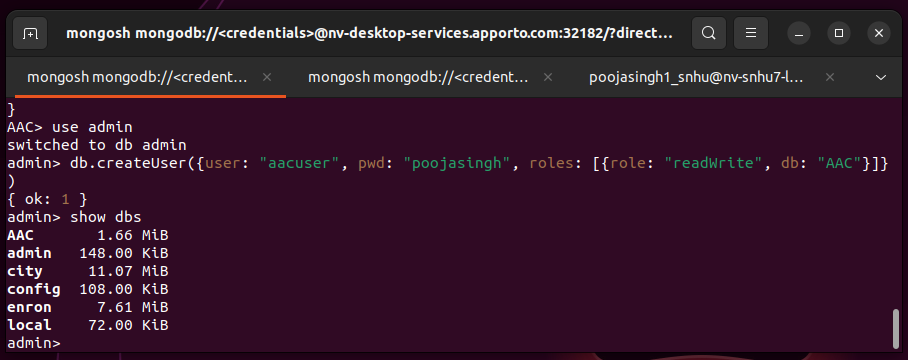
If this project needed to be recreated due to server migration or failed back up the following steps can be followed.

1. Install the required components, see section *Installation/Pre-Conditions*
2. Use an admin account to load data into MongoDB, *mongoimport* is suggested

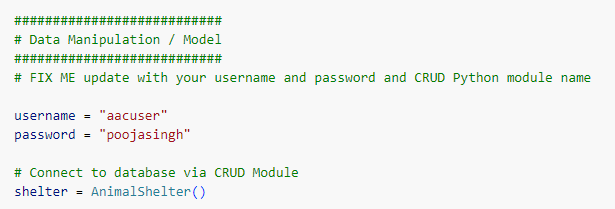


1. Create the needed user roles (CRUD class does not need full admin rights just read/write) – two accounts (admin and a user role) are shown below



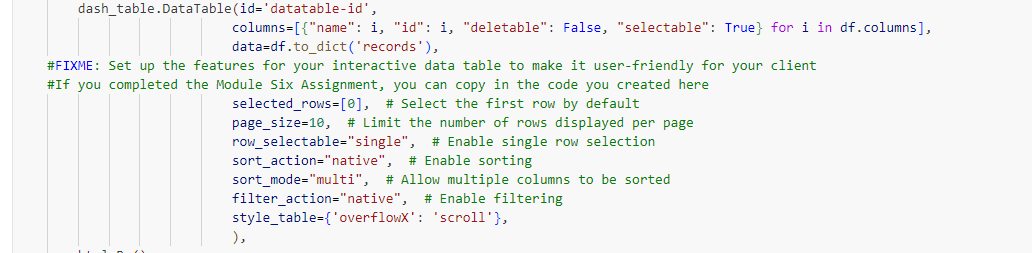


1. Add the CRUD class and the middleware-dashboard driver to the host server
2. Update the middleware-dashboard driver with username and password created in **Step 3**

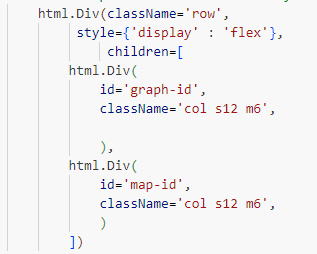


1. Add the Dash components and their callbacks (see Appendix) as needed

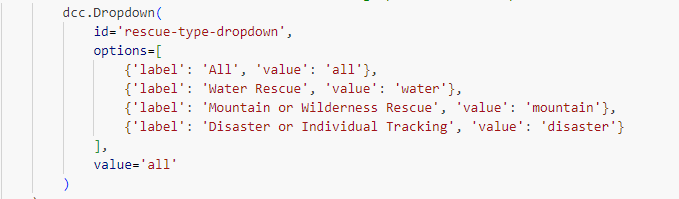
* Data table code example



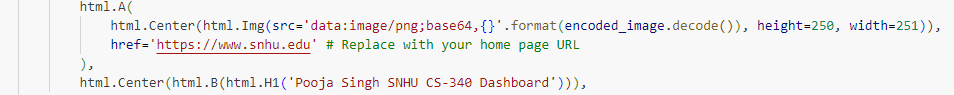
* Html div holding widgets example



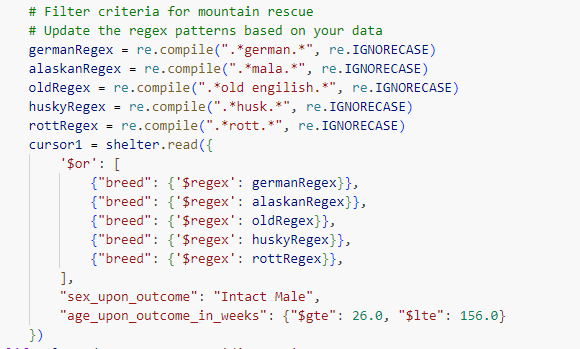
* Dropdown box acting as filters example



* Html anchor for logo and link to homepage example



1. Update any regular expressions used for filtering if they have changed, Mountain or Wilderness Rescue is shown here



1. Start MongoDB
2. Start the Python middleware-dashboard driver

# Pit Falls and Struggles

When recreating the project, many encountered challenges filtering the Austin Animal Shelter data due to inconsistencies in the breed names. For instance, Grazioso Salvare sought Chesapeake Bay Retrievers, but the data listed them as "Chesa Bay Retr." To address this, pattern matching using regular expressions was necessary. However, Pymongo doesn't directly accept regular expressions, requiring them to be formatted differently for compatibility with the API.

## Contact

Pooja Singh

# Appendix

# Dash callbacks

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

# Interaction Between Components / Controller

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

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

              [Input('rescue-type-dropdown', 'value')])

def update\_dashboard(selected\_rescue\_type):

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

    if selected\_rescue\_type == 'all':

        cursor1 = shelter.read({})

    elif selected\_rescue\_type == 'water':

        # Filter criteria for water rescue

        # Update the regex patterns based on your data

        labRegex = re.compile(".\*lab.\*", re.IGNORECASE)

        chesaRegex = re.compile(".\*chesa.\*", re.IGNORECASE)

        newRegex = re.compile(".\*newf.\*", re.IGNORECASE)

        cursor1 = shelter.read({

            '$or': [

                {"breed": {'$regex': newRegex}},

                {"breed": {'$regex': chesaRegex}},

                {"breed": {'$regex': labRegex}},

            ],

            "sex\_upon\_outcome": "Intact Female",

            "age\_upon\_outcome\_in\_weeks": {"$gte": 26.0, "$lte": 156.0}

        })

    elif selected\_rescue\_type == 'mountain':

        # Filter criteria for mountain rescue

        # Update the regex patterns based on your data

        germanRegex = re.compile(".\*german.\*", re.IGNORECASE)

        alaskanRegex = re.compile(".\*mala.\*", re.IGNORECASE)

        oldRegex = re.compile(".\*old engilish.\*", re.IGNORECASE)

        huskyRegex = re.compile(".\*husk.\*", re.IGNORECASE)

        rottRegex = re.compile(".\*rott.\*", re.IGNORECASE)

        cursor1 = shelter.read({

            '$or': [

                {"breed": {'$regex': germanRegex}},

                {"breed": {'$regex': alaskanRegex}},

                {"breed": {'$regex': oldRegex}},

                {"breed": {'$regex': huskyRegex}},

                {"breed": {'$regex': rottRegex}},

            ],

            "sex\_upon\_outcome": "Intact Male",

            "age\_upon\_outcome\_in\_weeks": {"$gte": 26.0, "$lte": 156.0}

        })

    elif selected\_rescue\_type == 'disaster':

        # Filter criteria for disaster rescue

        # Update the regex patterns based on your data

        germanRegex = re.compile(".\*german.\*", re.IGNORECASE)

        goldenRegex = re.compile(".\*golden.\*", re.IGNORECASE)

        bloodRegex = re.compile(".\*blood.\*", re.IGNORECASE)

        doberRegex = re.compile(".\*dober.\*", re.IGNORECASE)

        rottRegex = re.compile(".\*rott.\*", re.IGNORECASE)

        cursor1 = shelter.read({

            '$or': [

                {"breed": {'$regex': germanRegex}},

                {"breed": {'$regex': goldenRegex}},

                {"breed": {'$regex': bloodRegex}},

                {"breed": {'$regex': doberRegex}},

                {"breed": {'$regex': rottRegex}},

            ],

            "sex\_upon\_outcome": "Intact Male",

            "age\_upon\_outcome\_in\_weeks": {"$gte": 20.0, "$lte": 300.0}

        })

    else:

        raise Exception("Unknown filter")

    cursor1.rewind()

    df\_filtered = pd.DataFrame.from\_records(cursor1)

    data = df\_filtered.to\_dict('records')

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

    return data

# Display the breeds of animal based on quantity represented in

# the data table

@app.callback(

    Output('graph-id', "children"),

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

def update\_graphs(viewData):

    ###FIX ME ####

    # add code for chart of your choice (e.g. pie chart) #

    if viewData is None or len(viewData) == 0:  # Check if viewData is None or empty

        return "No data available"

    dffPie = pd.DataFrame.from\_dict(viewData)

    return [

        dcc.Graph(

            figure = px.pie(dffPie, names='breed', title='Preferred Animals')

        )

    ]

#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):

    if selected\_columns is None:

        return []  # Return an empty list if selected\_columns is None

    else:

        return [{

            'if': { 'column\_id': i },

            'background\_color': '#D2F3FF'

        } for i in selected\_columns]

# 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):

    if viewData is None:

        return

    elif index is None:

        return

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

    # Because we only allow single row selection, the list can be converted to a row index here

    if viewData is None or len(viewData) == 0:

        return "No data available"  # Handle case where there is no data

    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])

                ])

            ])

        ])

    ]