# CS 340 README for Grazioso Salvare’s Custom Dashboard

# About the Project

This project was conducted to build a functional, web-based, dashboard for Grazioso Salvare so they may review animals available at the Austin Animal Shelter that fit their business needs.

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

The Python middleware is using a custom class that facilitates Create-Read-Update-Delete (CRUD) actions in MongoDB via the Pymongo driver. The class will contain all the abstracted getter/setter methods needed for CRUD operations and MongoDB initiation.

# Component Selection Methodology

MongoDB was chosen over SQL due to the ease of use and its flexibility. MongoDB uses a dynamic schema which helps to prevent errors occurring during an update or create operation, i.e., someone enters an age with decimals, but it was typed to an integer. Moreover, the JSON format from MongoDB is better suited to a web environment.

MongoDB does not guarantee the same level of ACID properties as SQL, but this tool is not expected to have many transactions so this issue can be disregarded.

Python was chosen since it has many tools available to connect the backend and front end. For example, the Pymongo driver is used to interact with MongoDB via the CRUD class and middle-ware layer. Python can also build html/JavaScript based dashboards and other widgets via the Plotly-Dash, Dash Leaflet libraries.

Plotly-Dash, Dash Leaflet libraries are wrappers around JavaScript based tools that interact with Python through their respective APIs. These libraries make it possible to generate the SVG (pie chart), the tiled-PNG (map), or generate the raw html code and CSS needed to render the data table and widgets.

#### CRUD Class

CRUD operations are commonplace when using any database but formatting these queries and any error handling can be tedious. To help alleviate the tedium of formatting each new query or re-code for each error type these operations have been abstracted to simple get and set methods.

The class contains the following methods:

* Constructor (password, username)
* createRecord(data)
* deleteRecord (query)
* getRecordId(data)
* getRecordCriteria(criteria = None)
* updateRecord (query, newValue)

And properties:

* records\_updated
* records\_matched
* records\_deleted

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

# Installation/Pre-Conditions

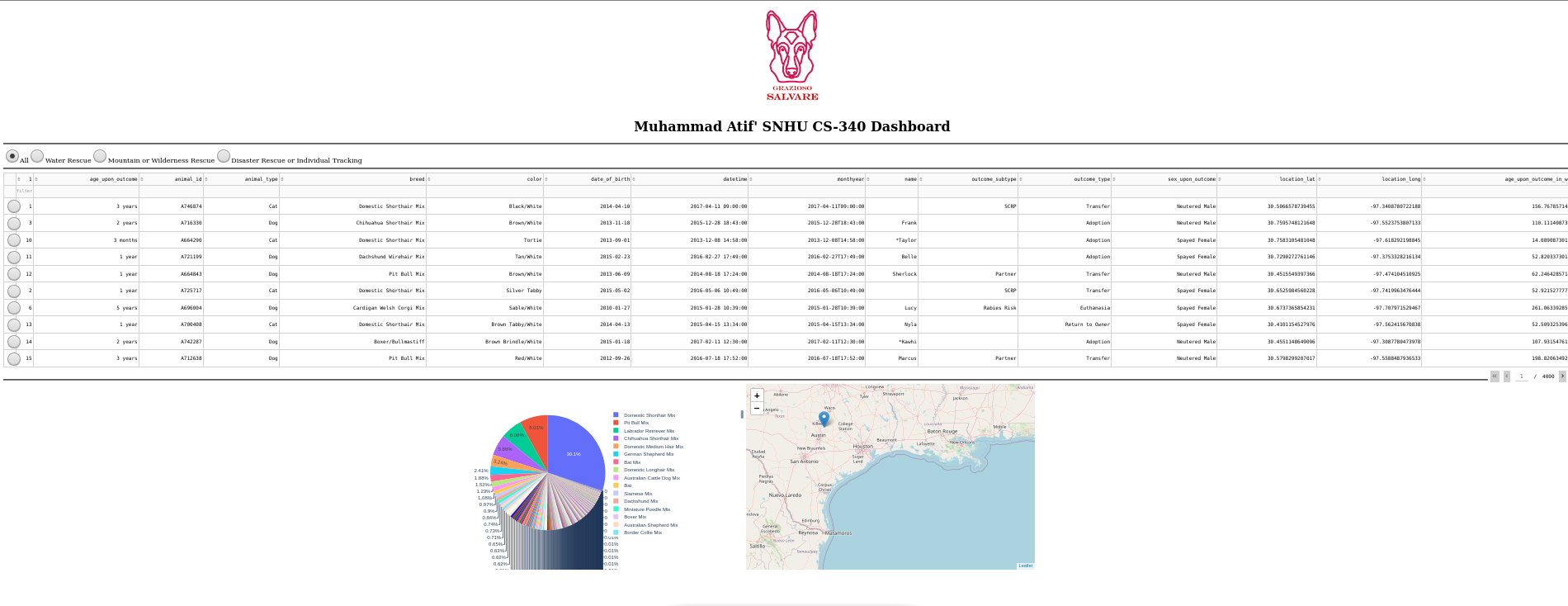
To use this software the following tools must be installed:

* Python 3.6
* Pymongo 4.2: [link](https://pymongo.readthedocs.io/en/stable/)
* MongoDB 4.2: [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



Toggle to control map

Inline filters and sorting available

Custom filters prebuilt for common searches

Dynamic pie chart showing available breeds

Map showing location of any selected animal

Data table that dynamically responds to user input

Client Logo also serving as a link to their homepage

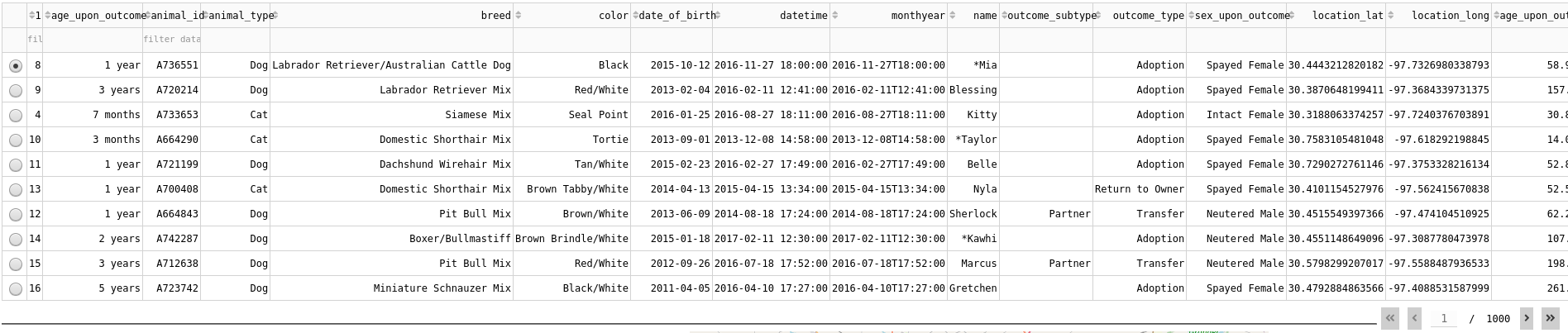
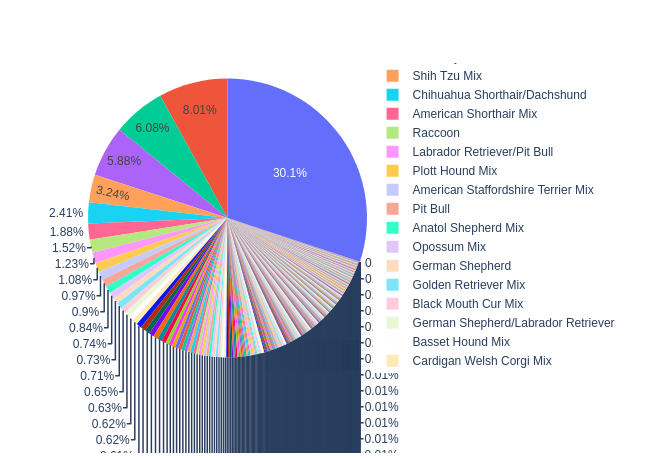
The custom filters are set with the specifications provided by the client, see below.

Table

Description automatically generated

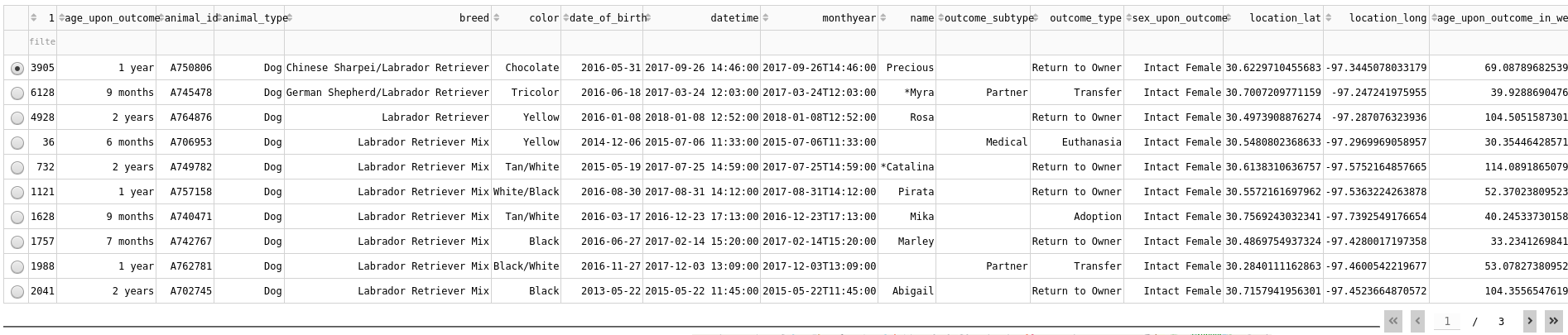
If a user chooses a new filter the pie chart and data table are updated, see below.





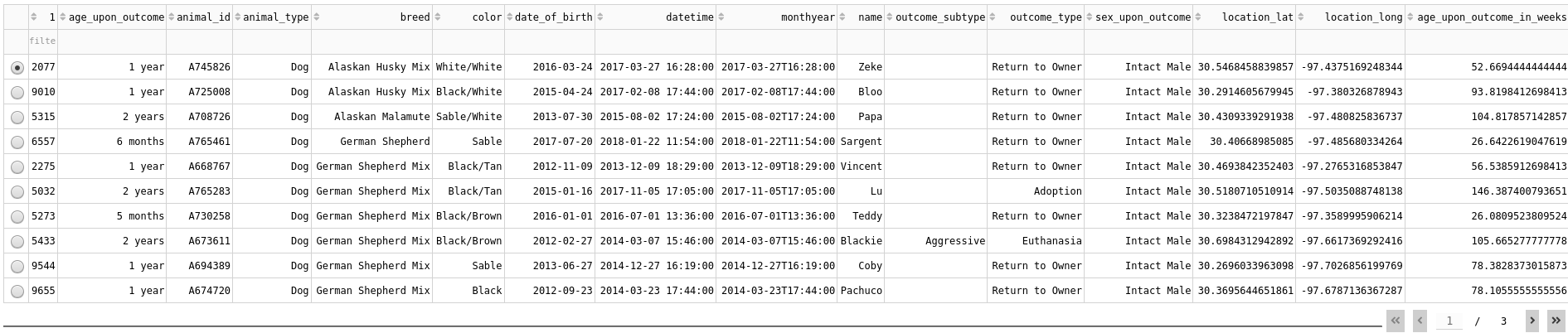


Chart, pie chart

Description automatically generated

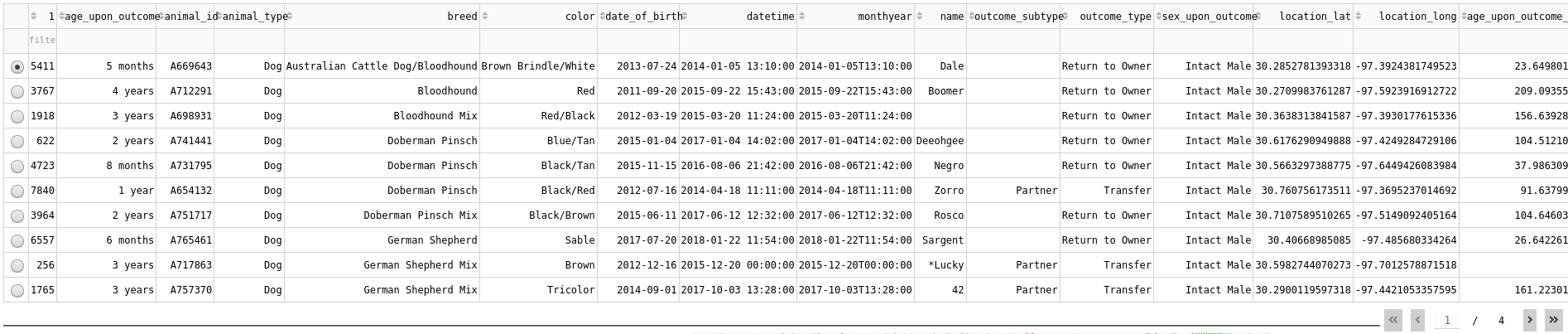


Chart, pie chart

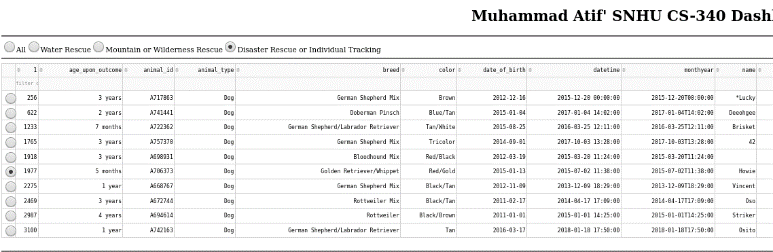
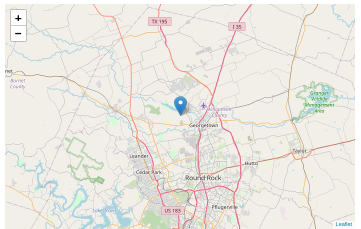
Description automatically generated



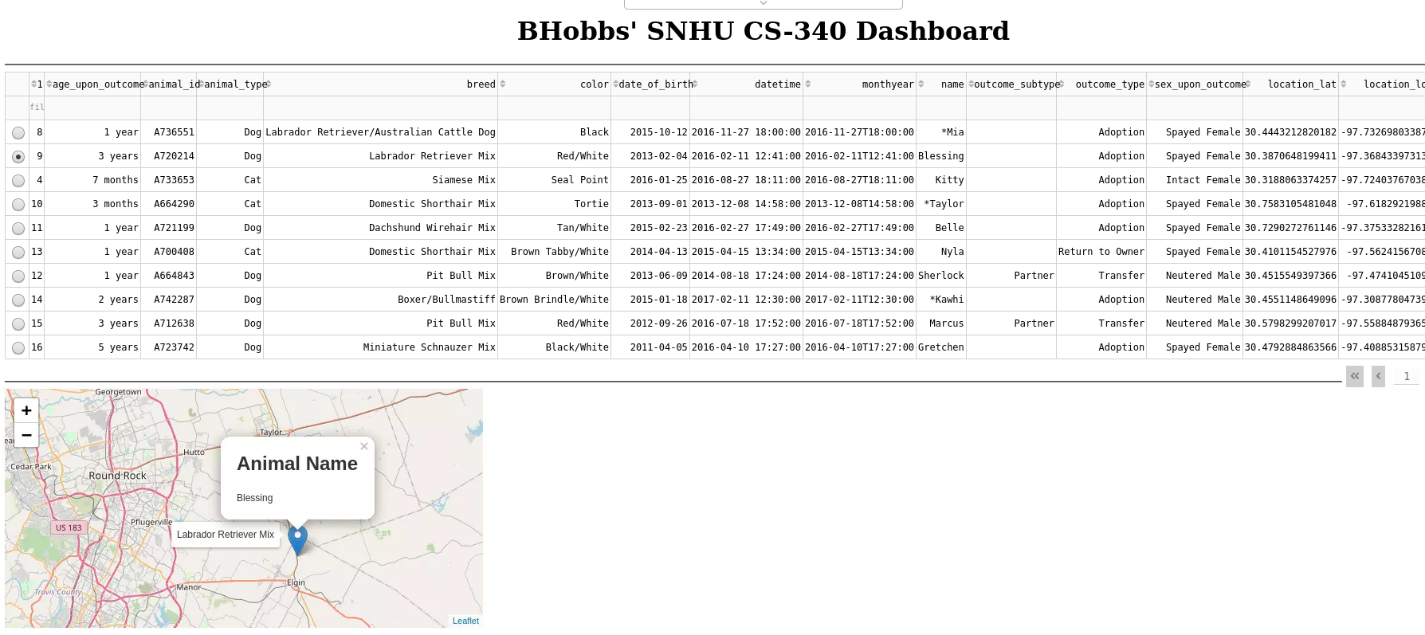
Chart, pie chart

Description automatically generated

If a particular animal is selected its location is shown on the map, otherwise the Austin Animal Shelter is shown, see below.

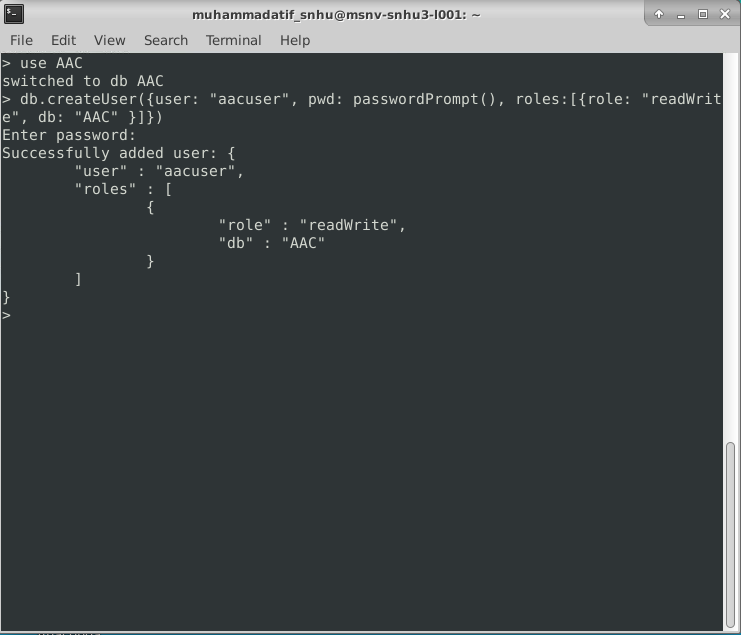
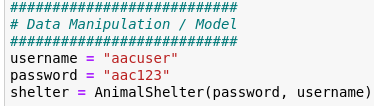
If the map marker is clicked the animal’s name is show and the breed is shown on hover.



# Project Recreation

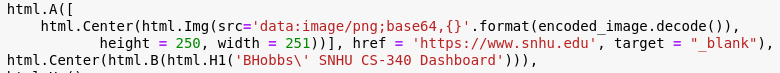
If this project needed to be recreated due to server migration or failed back up the following steps can be followed.

1. Install the required components, see section *Installation/Pre-Conditions*
2. Use an admin account to load data into MongoDB, *mongoimport* is suggested  
   
3. 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  
   Text

   Description automatically generated  
     
   
4. Add the CRUD class and the middleware-dashboard driver to the host server
5. Update the middleware-dashboard driver with username and password created in **Step 3**  
   
6. Add the Dash components and their callbacks (see Appendix) as needed
   * Data table code example  
     Graphical user interface, text, application

     Description automatically generated
   * Html div holding widgets example  
     Graphical user interface, text, application

     Description automatically generated
   * Radio buttons acting as filters example  
     A picture containing company name

     Description automatically generated
   * Html anchor for logo and link to homepage example  
     
7. Update any regular expressions used for filtering if they have changed, *Water Rescue* is shown  
   Text

   Description automatically generated
8. Start MongoDB
9. Start the Python middleware-dashboard driver

# Pit Falls and Struggles

For those trying to recreate the project the main struggle was filtering the Austin Animal Shelter data. Grazioso Salvare had specific requirements on the breeds of dogs but the data in the database is not very clean. For example, Grazioso Salvare is interested in the Chesapeake Bay Retriever breed, but this is listed as *Chesa Bay Retr* in the raw data. This means that pattern matching was needed to return a proper list of animals. Regular Expressions were employed for this pattern matching. Pymongo does not accept regular expressions, natively, and needed to be formatted differently for the Pymongo API to accept them.

## Contact

Muhammad Atif, [m.atif@snhu.edu](mailto:m.atif@snhu.edu),

# Appendix

# Dash callbacks

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

# Interaction Between Components / Controller

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

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

*Output('datatable-id','columns')],*

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

*def update\_dashboard(filter\_type):*

*#set up an if/else if/else block to respond to the radio buttons*

*if filter\_type == 'All':*

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

*elif filter\_type == 'Water':*

*#data isn't that clean, use regex for pattern matching*

*#build the regex patterns for the different filters*

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

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

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

*df = pd.DataFrame.from\_records(shelter.getRecordCriteria({*

*'$or':[ #Regex isn't allowed in an $in helper so use $or*

*{\"breed\": {'$regex': newRegex}}, #pass the regex to the filter*

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

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

*],*

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

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

*}))*

*elif filter\_type == 'Mountain':*

*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)*

*df = pd.DataFrame.from\_records(shelter.getRecordCriteria({*

*'$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 filter\_type == 'Disaster':*

*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)*

*df = pd.DataFrame.from\_records(shelter.getRecordCriteria({*

*'$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\")*

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

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

*return (data,columns)*

*#change the color of a selected cell*

*@app.callback(*

*Output('datatable-id', 'style\_data\_conditional'),*

*[Input('datatable-id', 'selected\_columns')]*

*)*

*def update\_styles(selected\_columns):*

*return [{*

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

*'background\_color': '#D2F3FF'*

*} for i in selected\_columns]*

*#call back for pie chart*

*#set to plot all of the data across all of the pages instead of the viewable data*

*#change to derived\_viewport\_data if other behavior is wanted*

*@app.callback(*

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

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

*def update\_graphs(viewData):*

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

*return [*

*dcc.Graph(*

*figure = px.pie(dffPie, names='breed',)*

*)*

*]*

*#call back for slecting a row and then plotting the geomarker*

*@app.callback(*

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

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

*def update\_map(virtualRows):*

*#austin Texas is [30.75, -97.48]*

*#create the views*

*if not virtualRows: #build a default view if there are no selected lines*

*markerArray = (30.75,-97.48) #default marker at Austin Animal Shelter*

*toolTip = "Austin Animal Center"*

*popUpHeading = "Austin Animal Center"*

*popUpParagraph = "Shelter Home Location"*

*else: #build the contextual views based on the selection*

*dff = pd.DataFrame(df.iloc[virtualRows]) #convert the datatable to a dataframe*

*coordLat = float(dff['location\_lat'].to\_string().split()[1]) #strip out the lat*

*coordLong = float(dff['location\_long'].to\_string().split()[1]) #strip out the long*

*markerArray = (coordLat, coordLong) #build the array based on selection*

*toolTip = dff['breed']*

*popUpHeading = "Animal Name"*

*popUpParagraph = dff['name']*

*#return the map with a child marker*

*#marker is set to the values found in markerArray*

*#map centers/moves to view the new marker instead of holding a fixed center*

*return [dl.Map(style={'width': '700px', 'height': '450px'}, center=markerArray,*

*zoom=10, children=[dl.TileLayer(id=\"base-layer-id\"),*

*dl.Marker(position=markerArray, children=[*

*dl.Tooltip(toolTip),*

*dl.Popup([*

*html.H1(popUpHeading),*

*html.P(popUpParagraph)*

*])*

*])*

*])*

*]*