# Project Two README

## About the Project

This project is a client-facing web application dashboard that allows a rescue-animal training company to locate and filter through available animals in nearby shelters. The client will access a MongoDB server populated with data points from five separate shelters in the Austin, Texas area.

## Motivation

The reason behind this project is to build an interactive and informative dashboard for a rescue-animal training company to make qualified business decisions. The project will educate them through the client dashboard on all animal statistics and filter through the best qualifying criteria for different rescue types.

## Getting Started

To replicate this project, the initial step is building out your database. It is recommended to use a MongoDB database instead of a SQL-based one due to the techniques and frameworks used. If you are unfamiliar with how to set up a MongoDB server you can reference their [installation page](https://www.mongodb.com/docs/manual/administration/install-community/) to get started.

The next step is ensuring your system has all the Python modules/libraries that you will need to make the dashboard function. See the Client-Side Installation list for specifics. [Anaconda](https://docs.anaconda.com/anaconda/install/) is a great distribution platform to manage the package management and deployment of this project. I will list all imports used in this project below to help understand acronyms and libraries being called. The final setup for the project is to import your CRUD functionality Python script and pull and read a query of all data in the collection you are working with. It should be able to properly create a class that connects to your server and reads queries successfully. This project does not focus on the create, update, and delete functionality.

from jupyter\_plotly\_dash import JupyterDash

import dash

import dash\_leaflet as dl

import dash\_core\_components as dcc

import dash\_html\_components as html

import plotly.express as px

import dash\_table as dt

from dash.dependencies import Input, Output, State

import os

import numpy as np

import pandas as pd

from pymongo import MongoClient

from bson.json\_util import dumps

import base64

from crud import AnimalShelter

The next section is creating and setting up the dashboard. We are using [Plotly Dash](https://dash.plotly.com/installation) for this project. The initial setup is initializing our class to run the app and then setting up the global variables we will use throughout the project. See the code example below to get an idea of what is needed.

# Creating our data frame

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

app = JupyterDash('SimpleExample')

# Image variables

image\_filename = 'grazioso\_salvare\_logo.png' # replace with your own image

encoded\_image = base64.b64encode(open(image\_filename, 'rb').read())

### Creating App Layout

Now we can create the visual side of the dashboard. Since Dash is browser-based, we will be using HTML-like terms and tags. Most of your layout will be separated into headings, like html.H1(), or into divisions (sections) like html.div(). You will find most HTML tags and how to call them within the Dash framework on their [HTML Components](https://dash.plotly.com/dash-html-components) documentation page. Refer to this page while you are creating the beginning of your layout.

Once you have a header, image, and any other elements of your choosing done, we can move on to creating your interactive data table and the remaining charts. In our project, we want to update our data table based on certain search criteria. We will do that with the use of radio buttons, checkboxes, or a dropdown menu. The example below uses radio buttons but the format is the same regardless of your choice.

html.Div(

dcc.RadioItems(

id='filter-type',

options=[

{'label': 'Water Rescue', 'value': 'wr'},

{'label': 'Mountain Rescue', 'value': 'mr'},

{'label': 'Disaster Rescue', 'value': 'dr'},

{'label': 'Reset', 'value': 'reset'},

],

)),

All “id” variables in app.layout like id='filter-type', will be referenced later when we make our callbacks. Creating a data table is as easy as calling a function with only two arguments. See the code below for the simple version.

dt.DataTable(df.to\_dict('records'), [{"name": i, "id": i} for i in df.columns])

While this will just populate the data table and provide nothing else, we can add customizations with additional code and arguments to the DataTable function. We can make the table editable, and sortable, delete rows and columns, as well as create pagination. Look at Plotly’s [Dash DataTable](https://dash.plotly.com/datatable) documentation for more information. The code below will be used for the customization options for this specific project.

dt.DataTable(

id='datatable-id',

columns=[

{"name": i, "id": i, "deletable": False, "selectable": True} for i

in df.columns],

],

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

# Additional customization options

editable=False,

sort\_action="native",

sort\_mode="multi",

column\_selectable=False,

row\_selectable=True,

row\_deletable=False,

selected\_columns=[],

selected\_rows=[],

page\_action="native",

page\_current= 0,

page\_size= 10,

),

The final layout section for this project will be a pie chart and a geolocation map to populate based on our above data table. This will involve little code in the app.layout section since most of the process is handled through our callback functions. Refer to the code below to set your graphs up in the HTML format.

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',)

]

)

### Creating Callbacks

We can now move on to the callback section of our dashboard. Using callbacks creates the functionality of our app. They will allow us to edit or update the data table and generate our pie chart and geolocation map. Refer to [Part 2 of Dash Fundamentals](https://dash.plotly.com/basic-callbacks) to better understand the basics of callbacks within this framework. Every callback starts with a specific statement. It will have an Output() and Input() argument being called to tell the callback where data is coming from and where to send it after your function or action is complete. This is where the “id” variables come into play. An example template is listed below to review.

@app.callback(

Output(component\_id='my-output', component\_property='children'),

Input(component\_id='my-input', component\_property='value')

)

Since we want to update our data table based on the selection of choices created above through radio buttons or whatever you chose, we will create our first callback. If you refer to the data table we created, we have our output id as id='datatable-id' with two variables of columns and data. Our input will reflect the id we used while creating our selection buttons, id='filter-type' and 'value’ which will pull the key listed afterward to use within our function as the input. Our specific callback will look like this:

@app.callback(

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

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

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

)

Now we can create the functionality of this callback. We will start with defining a function with the argument as filter\_type. This will allow us to access 'value' from our Input() argument in the callback. You should create your function to populate your MongoDB queries based on the criteria you selected earlier. You will populate your query within a new data frame. To avoid overwriting your df variable it is recommended to use an if-else statement.

def update\_dashboard(filter\_type):

# Water Rescue filter

if filter\_type == 'wr':

df = pd.DataFrame.from\_records(shelter.read({# Insert Query}))

# Mountain filter

elif filter\_type == 'mr':

df = pd.DataFrame.from\_records(shelter.read({# Insert Query}))

# Disaster Rescue filter

elif filter\_type == 'dr':

df = pd.DataFrame.from\_records(shelter.read({# Insert Query}))

# Reset to no filter

else:

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

After creating the function how you would like, you want to recreate the arguments used when initializing your data table and return both variables, columns and data into a tuple.

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

i in df.columns]

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

return (data,columns)

The next callback needed will handle creating our pie chart. We used Plotly’s [Pie Chart](https://plotly.com/python/pie-charts/) documentation to guide us along while creating this function. We can set up our pie chart by creating our callback, defining our function, and creating some local variables to use. Use the example below as a reference.

@app.callback(

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

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

def update\_graphs(viewData):

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

# assign the columns of our data that we will be using for legend names and values to calculate

names = dff['breed'].value\_counts().keys().tolist()

values = dff['breed'].value\_counts().tolist()

Now we can start our return statement which will generate our pie chart based on the data gathered above. We will be using two different libraries in this process. The first one will be dash\_core\_components for graph placement and plotly.express to create the pie chart. The plotly.express class has a method called .pie() which will take our functions variables and some styling options as arguments.

return [

dcc.Graph(

figure = px.pie(dff, values=values, names=names, title='Available Dogs by Breed', width=1000, height=500)

)

]

Our final callback will create and populate the geolocation map based on the main data table used. Again, we start it off with the proper @app.callback decorator and begin defining our function. We will need to use the same viewData parameter for this function and then create another local data frame.

@app.callback(

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

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

def update\_map(viewData):

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

pos\_curser = [dff.iloc[0,13], dff.iloc[0,14]]

You don’t have to create the pos\_curser variable. This was added to shorten some of the needed code in the upcoming return statement. Within our return statement, we will be accessing a plug-in called [Dash Leaflet](https://dash-leaflet-docs.onrender.com/) to create and modify our geolocation map. We start by calling the dl.Map() method and begin with listing our arguments to style the object. Next, we will begin building the functionality of the map. We will be using the class member's dl.TileLayer(), dl.Marker(), and dl.Tooltip() to position the map marker and provide more information on the data at that specific location. As you saw in the example above, we can access our data table locations by using the syntax, dff.iloc[#row\_number,#col\_number].

return [

dl.Map(style={'width': '1000px', 'height': '500px'},

center=pos\_curser, zoom=10, children=[

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

# Marker with tooltip and popup

dl.Marker(position=pos\_curser, children=[

dl.Tooltip(dff.iloc[0,4]),

dl.Popup([

html.H1('Animal Name'),

html.p(dff.iloc[0,9])

])

])

])

]

The final job to do is run the app. This is simply done by putting app as the last statement at the very end of your program.

app = JupyterDash('App Name')

app.layout = html.Div(# Layout code in here)

@app.callbacks()

def callback\_functions()

# Code

app

## Installation

Server Side:

* MongoDB
* Linux/Windows OS

Client Side:

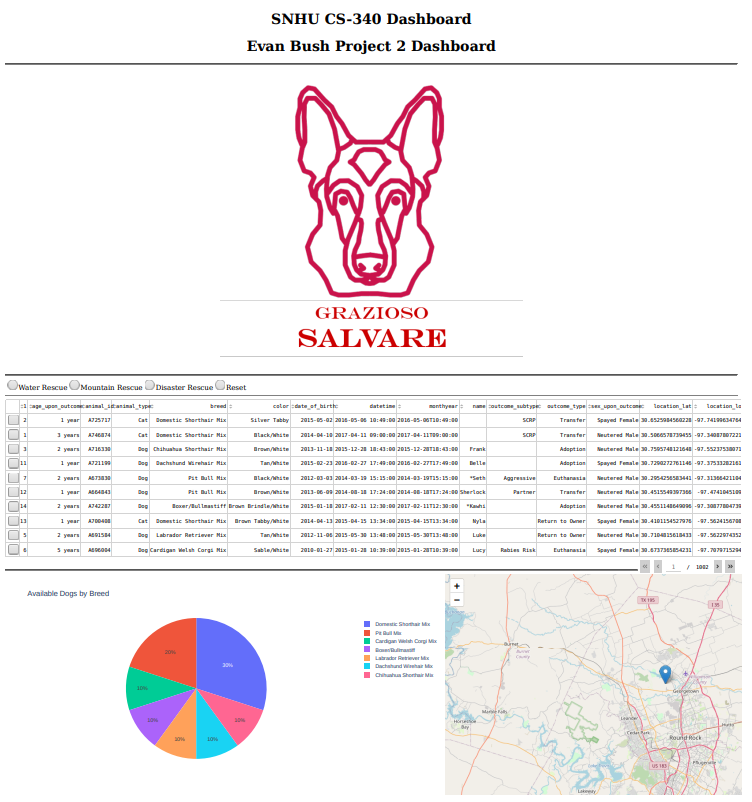
* Python: Anaconda distribution
* Python Modules/Libraries: jupyter\_plotly\_dash, dash, dash\_leaflet, dash\_core\_components, dash\_html\_components, plotly.express, dash\_table, dash.dependencies, os, numpy, pandas, pymongo, bson.json\_util, base64

## Usage

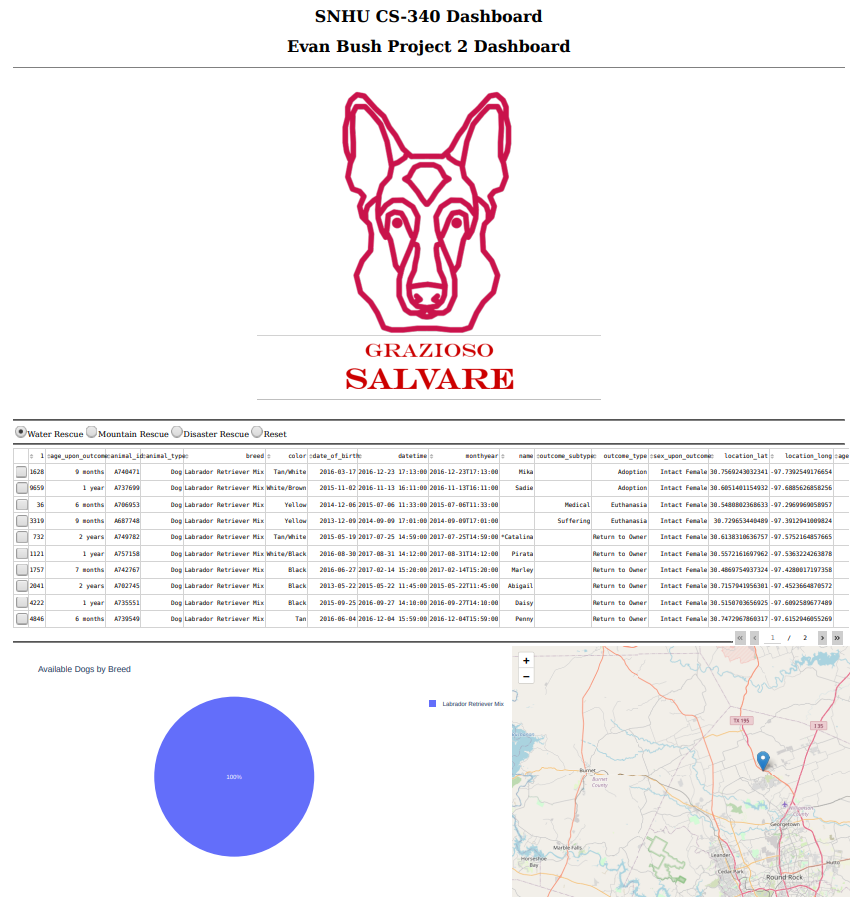
Each section from the app layout to the callbacks are all designed to make the dashboard visually appealing and functional. With proper set up you can create a dashboard that will allow you to sort and filter data as needed, update and delete documents from your database, and even visualize the data in an effective manner. Whether it’s maintaining information or using data to make important decisions, this project can help you or your client achieve the necessary goals established.

If you want to make sure your app is running as intended, when you start the program, you can test out the different filtering options to see if the data table pulls the correct information and the charts update accordingly as well. Refer to the screenshots below to guide in what the result should look like for each of the guided filter options.

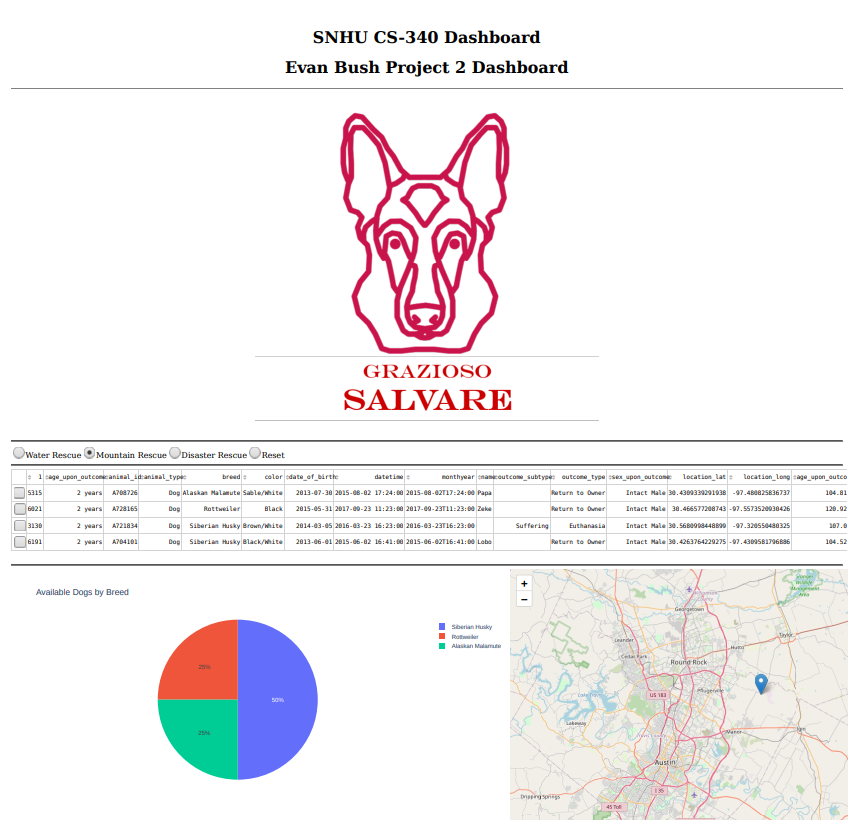
### Original State



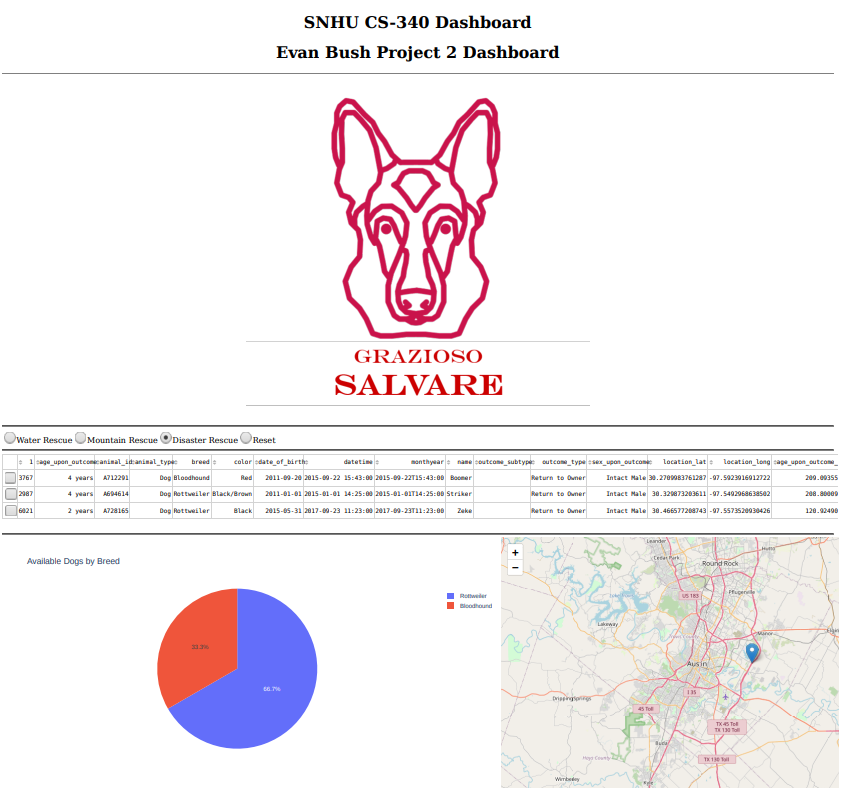
### Water Rescue State



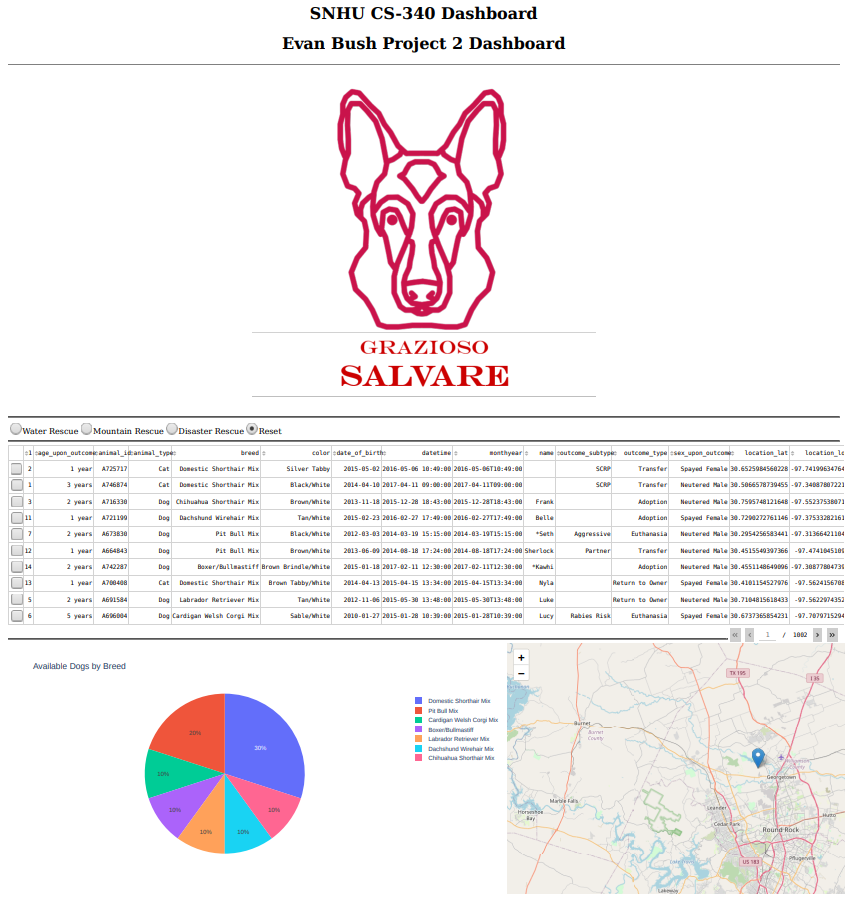
### Mountain Rescue State



### Disaster Rescue State



### Reset State



## Contact

Evan Bush