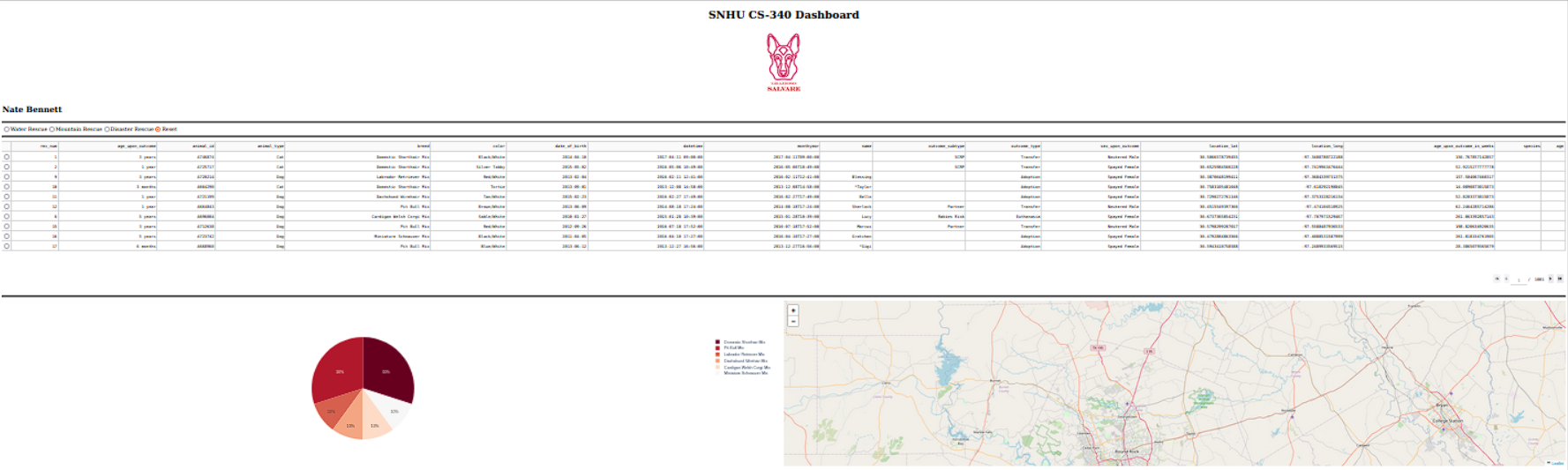
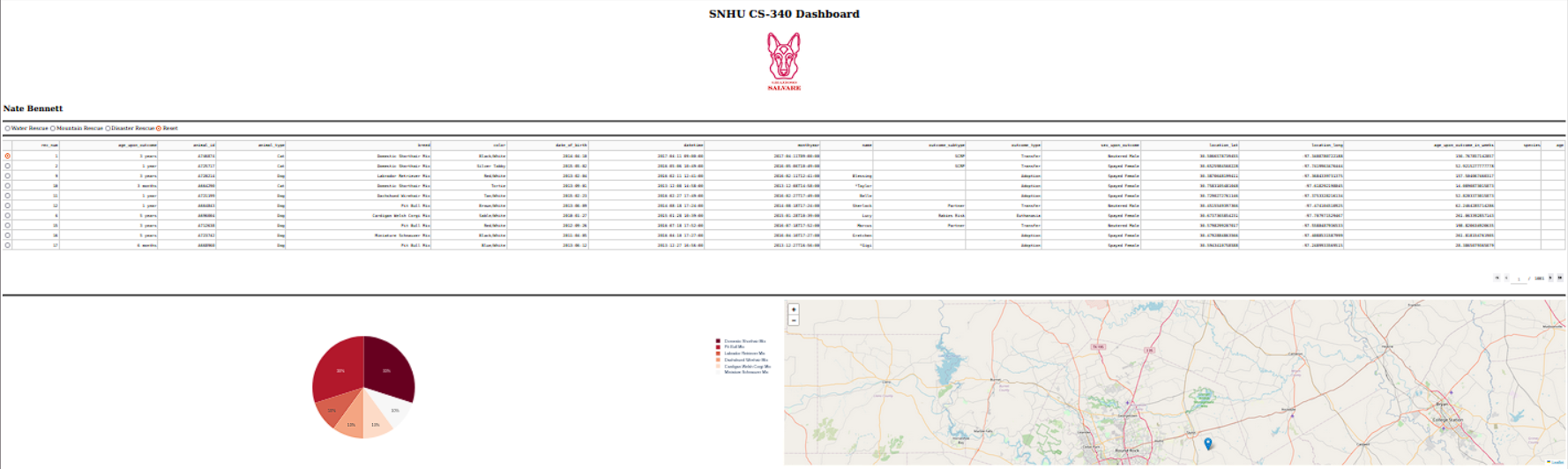
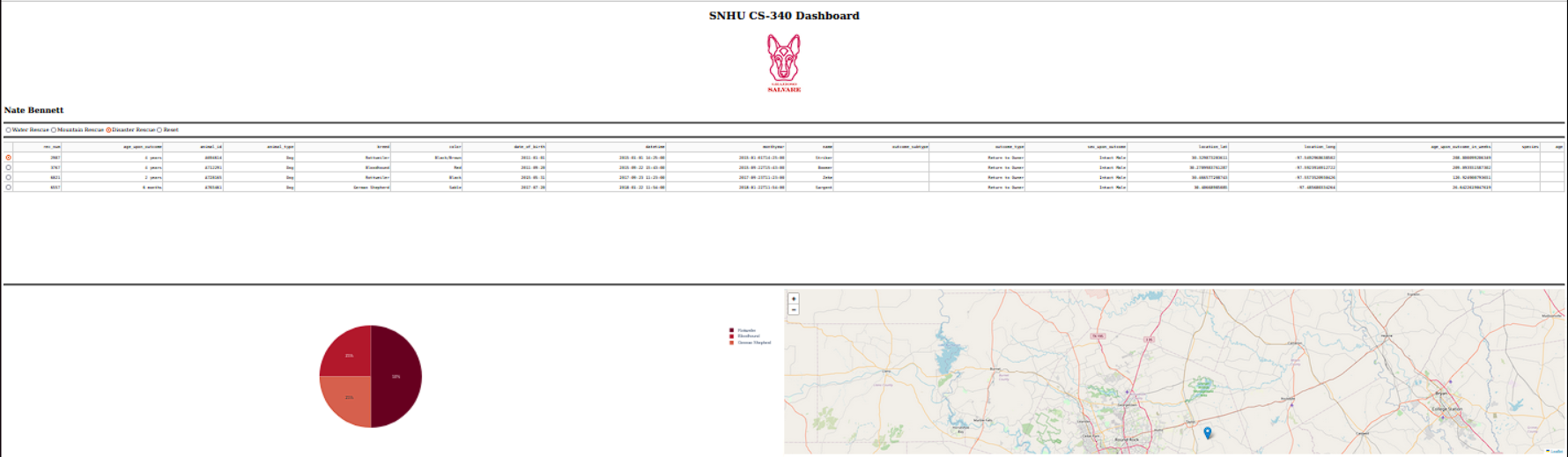
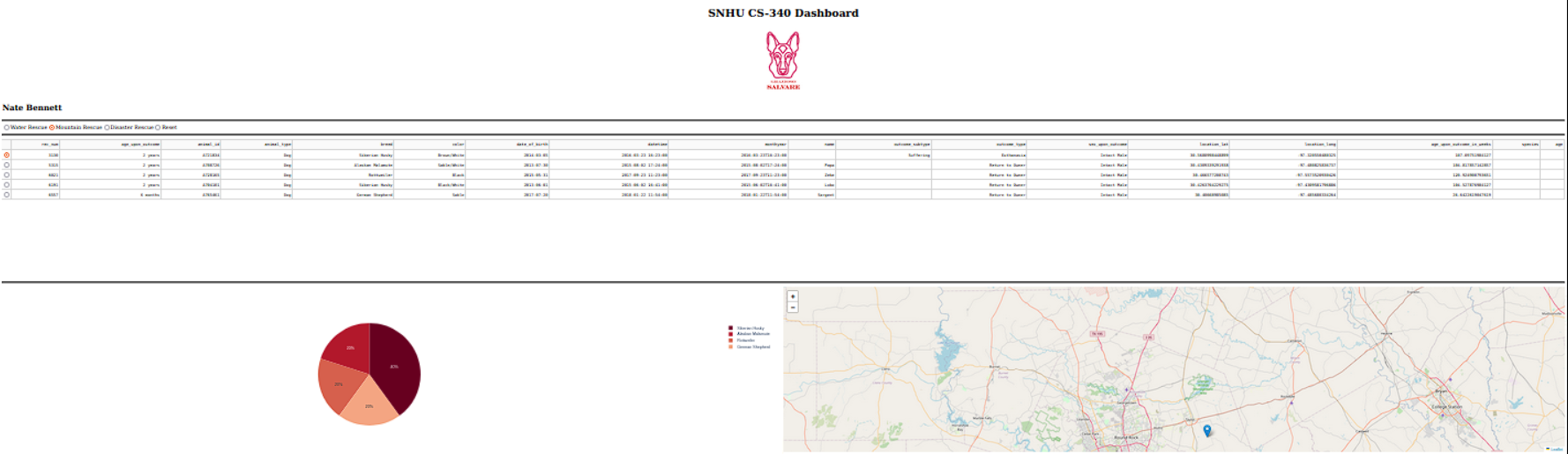
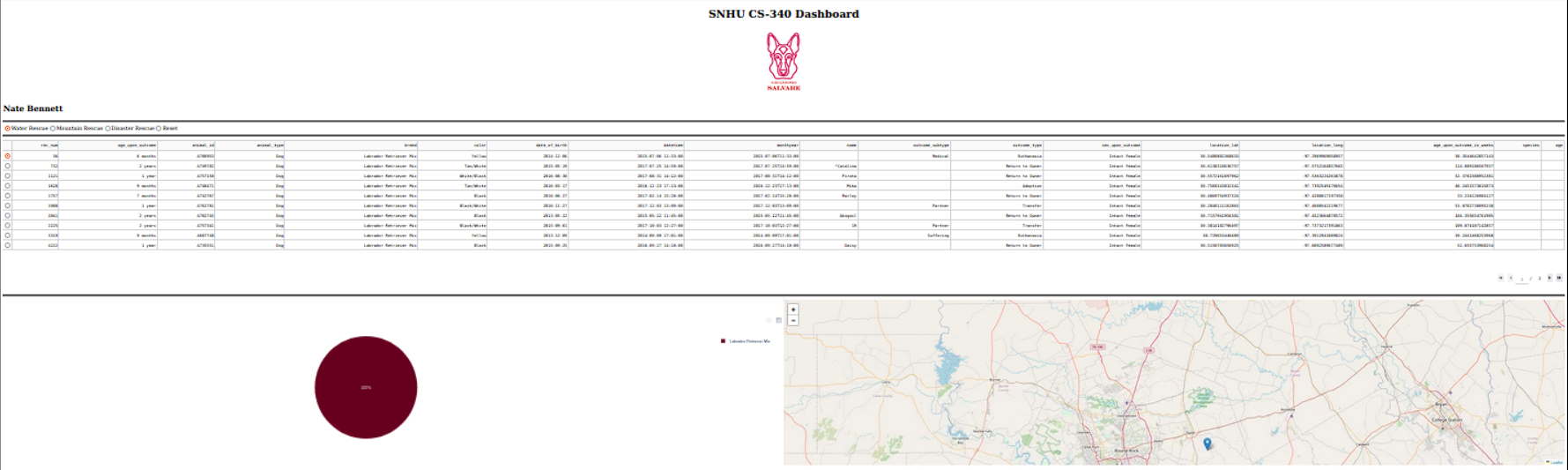
**Nate Bennett**

**7-2 Project Two README File**

**Required Functionality**

This is a web application project built using the dash framework, a MongoDB database, and a Python script that displays dog data. The program has filters to narrow the dog data down to see what dogs might be decent candidates for search and rescue training. The dashboard has a map that displays the selected dog's latitude and longitude. The program also has a pie chart that displays the percentage of each breed found in the data set.



Some crud operations.

def \_\_init\_\_(self, db\_name, collection\_name, user, password, host, port):

# Initialize the MongoClient using the credentials from environment variables.

mongo\_uri = f"mongodb://{user}:{password}@{host}:{port}/{db\_name}?authSource=admin&authMechanism=SCRAM-SHA-256"

self.client = MongoClient(mongo\_uri)

self.database = self.client[db\_name]

self.collection = self.database[collection\_name]

def create(self, data):

""" This should insert a document into the specified MongoDB database and collection. """

if data:

result = self.collection.insert\_one(data)

return True if result.inserted\_id else False

else:

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

def read(self, query=None):

""" This should query for documents from the specified MongoDB database and collection. """

if query is None:

cursor = self.collection.find({})

else:

cursor = self.collection.find(query)

result = [doc for doc in cursor]

return result

def update(self, query, update\_data, is\_multi=False):

""" This should update document(s) in the specified MongoDB database and collection. """

if query:

if is\_multi:

result = self.collection.update\_many(query, {"$set": update\_data})

else:

result = self.collection.update\_one(query, {"$set": update\_data})

return result.modified\_count

else:

raise Exception("Query parameter cannot be empty")

def delete(self, query, is\_multi=False):

""" This should delete one or more documents from the specified MongoDB database and collection. """

if query:

if is\_multi:

result = self.collection.delete\_many(query)

else:

result = self.collection.delete\_one(query)

return result.deleted\_count

else:

raise Exception("Query parameter cannot be empty")

Some filtering Examples.

dcc.RadioItems(

id='filter\_type',

options=[

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

{'label': 'Mountain Rescue', 'value': 'Mountain or Wilderness Rescue'},

{'label': 'Disaster Rescue', 'value': 'Disaster Rescue or Individual Tracking'},

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

],

value='Reset',

labelStyle={'display': 'inline'}

),

@app.callback(

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

[Input('filter\_type', 'value')])

def update\_dashboard(filter\_type):

if filter\_type == 'Reset':

# Reset button clicked, show all data

query={}

elif filter\_type == 'Water Rescue':

query = {

'$and': [

{'sex\_upon\_outcome': 'Intact Female'},

{'breed': {'$in': ['Labrador Retriever Mix', 'Chesa Bay Retr Mix', 'Newfoundland/Labrador Retriever',

'Newfoundland Mix', 'Newfoundland/Great Pyrenees',

'Newfoundland/Australian Cattle Dog', 'Labrador Retriever/Newfoundland']}},

{'age\_upon\_outcome\_in\_weeks': {'$gte': 26, '$lte': 156}}

]

}

elif filter\_type == 'Mountain or Wilderness Rescue':

query = {

'$and': [

{'sex\_upon\_outcome': 'Intact Male'},

{'breed': {'$in': ['German Shepherd', 'Alaskan Malamute', 'Old English Sheepdog',

'Rottweiler', 'Siberian Husky']}},

{'age\_upon\_outcome\_in\_weeks': {'$gte': 26, '$lte': 156}}

]

}

elif filter\_type == 'Disaster Rescue or Individual Tracking':

query = {

'$and': [

{'sex\_upon\_outcome': 'Intact Male'},

{'breed': {'$in': ['Doberman Pinscher', 'German Shepherd', 'Golden Retriever',

'Bloodhound', 'Rottweiler']}},

{'age\_upon\_outcome\_in\_weeks': {'$gte': 20, '$lte': 300}}

]

}

filtered\_df = pd.DataFrame.from\_records(crud.read(query))

# If '\_id' exists in the columns, drop it

if '\_id' in filtered\_df.columns:

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

return filtered\_df.to\_dict('records')

Some graph and map code.

@app.callback(

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

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

)

def update\_graphs(viewData):

# Create a DataFrame from the displayed data

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

# Get breed names and their counts for the pie chart

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

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

return [

dcc.Graph(

figure=px.pie(

data\_frame=dff,

values=values,

names=names,

color\_discrete\_sequence=px.colors.sequential.RdBu,

)

)

]

@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]

@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 index is None or len(index) == 0:

return dl.Map(

style={'width': '100%', 'height': '100%'},

center=[30.75, -97.48], zoom=10, # Default to Austin, TX

children=[

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

]

)

else:

row = index[0]

return dl.Map(

style={'width': '100%', 'height': '100%'},

center=[df.iloc[row, 13], df.iloc[row, 14]], zoom=10,

children=[

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

dl.Marker(

position=[df.iloc[row, 13], df.iloc[row, 14]],

children=[

dl.Tooltip(df.iloc[row, 4]),

dl.Popup([

html.H1("Animal Name"),

html.P(df.iloc[row, 9])

])

]

)

]

)

**Tools**

Some tools that were used in this application are Dash, which is used for building the application's user interface, Dash Leaflet which is used for rendering interactive maps, Plotly which is used for creating data visualizations and charts, MongoDB which acts as a database, and Jupyter Notebook to run the application.

MongoDB was chosen because of its flexibility and ability to store JSON files as well as its compatibility with Python and structured NoSQL data management.

The Dash framework was chosen due to its ability to help create data driven web applications. Dash provides both the front end view and back end controller components of a web application. The layout of the application is defined in a Python script using HTML and dash components and callbacks are used to respond to user interaction. Dash works with libraries like Plotly to create interactive data visualizations, and Dash can help provide components like tables, charts, and maps.

**Steps**

Some steps taken to create this project was to set up the development environment, like importing the MongoDB database, creating a database account, creating a CRUD Python script to help handle MongoDB commands like create, read, update, and delete, Setting up an interactive dashboard that utilized the CRUD file to provide an interactive data driven application that has a unique identifier, logo, table, filtering functionality, map, and chart.

**Challenge**

One potential development challenge was finding what port to run the application on when testing. I believe what I did was ran MongoDB in the terminal and found the port number.