Vinita Ravivarma

Dr. Kellogg

CS-340-12053-M01

February 23rd, 2025

7-2 Project Two Submission

```
# Setup the Jupyter version of Dash
from jupyter_dash import JupyterDash

# Configure the necessary Python module imports for dashboard components
import dash
import dash_leaflet as dl
from dash import dcc
from dash import html
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

#### FIX ME #####
# change animal_shelter and AnimalShelter to match your CRUD Python module file name and class name
from CRUD import AnimalShelter
```

```
username = "aacuser"
password = "SNHU1234"

# Connect to database via CRUD Module
shelter = 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({}))

# MongoDB v5+ is going to return the '_id' column and that is going to have an
# invlaid object type of 'ObjectID' - which will cause the data_table to crash - so we remove
# it in the dataframe here. The df.drop command allows us to drop the column. If we do not set
# inplace=True - it will reeturn a new dataframe that does not contain the dropped column(s)
df.drop(columns=['_id'],inplace=True)

## Debug
# print(len(df.to_dict(orient='records')))
# print(df.columns)

#########################
# Dashboard Layout / View
#########################
app = JupyterDash('SimpleExample')

#FIX ME Add in Grazioso Salvare's logo
image_filename = 'grazioso_salvare_logo.png' # replace with your own image
encoded_image = base64.b64encode(open(image_filename, 'rb').read())
```

```
app.layout = html.Div([
#    html.Div(id='hidden-div', style={'display':'none'}),
    html.Center(html.B(html.H1('CS-340 Dashboard'))),
    html.Hr(),
    html.Div(

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

    ),
    html.Hr(),
    dash_table.DataTable(id='datatable-id',
                         columns=[{"name": i, "id": i, "deletable": False, "selectable": True} for i in df.columns],
                         data=df.to_dict('records'),
#FIXME: Set up the features for your interactive data table to make it user-friendly for your client
#If you completed the Module Six Assignment, you can copy in the code you created here

                        ),
    html.Br(),
    html.Hr(),
#This sets up the dashboard so that your chart and your geolocation chart are side-by-side
    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',
            )

        ])
])

############################################
# Interaction Between Components / Controller
############################################



@app.callback(Output('datatable-id','data'),
              [Input('filter-type', 'value')])
def update_dashboard(filter_type):
## FIX ME Add code to filter interactive data table with MongoDB queries
#
#
#        columns=[{"name": i, "id": i, "deletable": False, "selectable": True} for i in df.columns]
#        data=df.to_dict('records')
#
#
#        return (data,columns)

# 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) #

    #return [
    #    dcc.Graph(
```

```
    #         figure = px.pie(df, 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):
    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 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=[30.75,-97.48], children=[
                dl.Tooltip(dff.iloc[0,4]),
                dl.Popup([
                    html.H1("Animal Name"),
                    html.P(dff.iloc[0, 10])
                    html.H3("Animal Color"),
                    html.P(dff.iloc[0, 5]),
                    html.H3("Animal ID"),
                    html.P(dff.iloc[0, 2]),
                ])
            ])
        ])
    ]


app.run_server(debug=True)
```

The code creates a dashboard using Python libraries such as Dash and Plotly. Data handling connects to a database of an animal shelter, fetches data, and displays it in a table format. For user interaction, users may filter the data according to different rescue types like mountain or water rescue. There are two ways to visualize data, which are the data table and map. However, the data table shows detailed information about animals in the shelter while a map demonstrates the location of selected animals there. Callbacks are these functions updating the dashboard based on user actions. According to the situation, the data table gets updated and only displays

relevant information when users choose a filter type. When users select columns in the data table, it highlights those columns. Therefore, it offers placeholders to update graphs and maps based on the selected data, which can be customized further. Launching the app initializes and displays the dashboard in a Jupyter notebook. That code also creates an interactive dashboard where users may explore and visualize data from an animal shelter database easily.