Dash App for Grazioso Salvare's Animal Records

# About the Project

A screenshot of a computer

Description automatically generated

Grazioso Salvare, a renowned animal shelter and rescue service, needed a digital transformation to manage their animal records. This CRUD Dash application serves to streamline the process of inputting, retrieving, updating, and deleting records for animals under the care of Grazioso Salvare, ensuring efficient and accurate data management.

# Motivation

The heartbeat of Grazioso Salvare is its commitment to the welfare and care of animals. As the number of animals under their care began to swell, the management was faced with an increasing challenge: handling a vast and dynamic database of animal records. Traditional methods and manual entries began to falter, creating inefficiencies, possible inaccuracies, and delay in vital tasks.

The primary goal was to bridge this gap. The vision was a platform that was not only efficient but also user-friendly, especially considering the non-technical background of many staff members. It was imperative to design a system that allowed easy input, modification, and retrieval of records without the need for extensive technical know-how.

This urgent need for enhanced data management, combined with the shelter's unwavering commitment to the animals, became the driving force behind this project. It wasn't just about digitization, but about providing the best tools for the staff to ensure the best care for the animals.

# Development Tools

The development of the CRUD Dash application for Grazioso Salvare employed a curated set of tools tailored to the needs of the project. Each tool was specifically chosen for its features and relevance to the tasks at hand. Here's a more in-depth exploration:

## 1. Python

Python, known for its simplicity and powerful libraries, was foundational to the entire project. It allowed for the creation of complex functionalities with concise code.

Example Code: Class Initialization from Project

class CRUD(object):

def \_\_init\_\_(self, USER='aacuser', PASS='aacPass', HOST='nv-desktop-services.apporto.com', PORT=32080, DB='AAC', COL='animals'):

# Initialize Connection

self.client = MongoClient('mongodb://%s:%s@%s:%d' % (USER, PASS, HOST, PORT))

self.database = self.client[DB]

self.collection = self.database[COL]

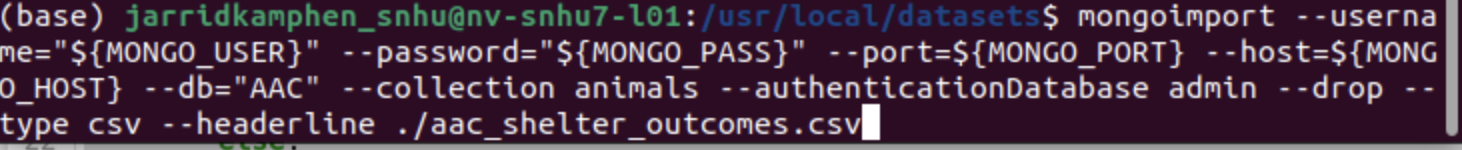
## 2. Dash

Dash, a Python framework, empowered the creation of the web interface for the CRUD operations. This framework was crucial in making the application user-friendly and accessible.

(Note: I don't have a direct Dash code from the provided text, but ideally, it would involve UI components and callback functions.)

## 3. MongoDB

MongoDB, a NoSQL database, provided the necessary flexibility to manage dynamic animal records. Its document-oriented structure was pivotal in efficiently storing and retrieving animal data.



def read(self, query):

"""Query documents from the collection."""

try:

result = list(self.collection.find(query))

return result

except Exception as e:

print("Error occurred:", e)

return []

## 4. PyMongo

PyMongo acted as the bridge, connecting Python with MongoDB. It translated Python function calls into MongoDB commands, streamlining the database operations.

# Getting Started

For those interested in setting up the project locally:

1. MongoDB: Ensure MongoDB is installed and running on your system.

2. Python Environment: It's recommended to set up a virtual environment. Install necessary   
libraries using `pip install -r requirements.txt` (ensure you're in the project directory).

3. Clone & Navigate: Clone the repository and navigate to the project folder.

4. Run the App: Execute the app using `python app.py`.

# Usage

With the Dash app:

1. \*\*Add New Animal Records\*\*: Intuitive forms allow easy input of new animal data, which is then saved to the MongoDB collection.

example:  
A screen shot of a computer program

Description automatically generated

2. \*\*Retrieve Animal Details\*\*: Using the search functionality, specific animal details can be retrieved based on different criteria.

example:

A screen shot of a computer code

Description automatically generated

3. \*\*Update & Delete\*\*: These functionalities will be incorporated in future updates, enabling complete CRUD operations.

A screen shot of a computer code

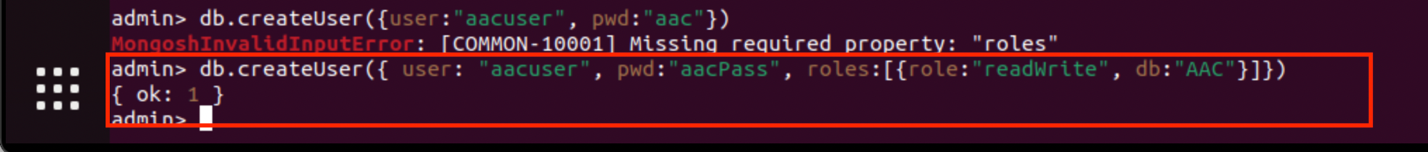
Description automatically generated

A screen shot of a computer code

Description automatically generated

# Milestones

- \*\*Database & User Authentication\*\*: For enhanced security, MongoDB was set up with user authentication. This ensures data integrity and restricts unauthorized access.

A screenshot of a computer

Description automatically generated

- \*\*Creation of C and R Functionality\*\*: The initial focus was on 'Create' and 'Read' operations. Challenges arose in integrating Dash with MongoDB, but with PyMongo, we were able to seamlessly link the two, leading to successful data input and retrieval.

Thank you for your interest in our CRUD Dash app for Grazioso Salvare. We continually strive for improvements and welcome feedback.

For those keen to explore the code:

# CODE

## Dash app:

from dash import html, dcc, dash\_table

from dash.dependencies import Input, Output, State

from CRUD\_module import CRUD

import pandas as pd

from jupyter\_dash import JupyterDash

import dash\_leaflet as dl

import plotly.express as px

# Database setup

username = "aacuser"

password = "aacPass"

db = CRUD(username, password)

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

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

# Dash app layout

app = JupyterDash('Grazioso Salvare')

app.layout = html.Div([

html.Center(html.B(html.H1('Grazioso Salvare'))),

html.Center(html.Img(src="assets/Grazioso Salvare Logo.png", height=100)),

html.Hr(),

# Rescue Type Filter

html.Div([

html.Label("Select Rescue Type"),

dcc.RadioItems(

id='rescue-type-radio',

options=[

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

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

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

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

],

value='all'

),

]),

# Dog Breed Dropdown

html.Div([

html.Label("Select Preferred Dog Breed"),

dcc.Dropdown(

id='dog-breed-dropdown',

options=[

{'label': label, 'value': label} for label in df[df['animal\_type'] == 'Dog']['breed'].unique()

],

value='all'

),

]),

html.Br(),

html.Button('Update', id='update-button'),

html.Div(id='test-output'),

# DataTable with highlighted row

dash\_table.DataTable(

id='table',

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

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

row\_selectable='single',

selected\_rows=[0],

page\_size=10,

style\_data\_conditional=[

{

'if': {'row\_index': 'odd'},

'backgroundColor': 'rgb(248, 248, 248)'

}

],

style\_header={

'backgroundColor': 'rgb(230, 230, 230)',

'fontWeight': 'bold'

}

),

# Map Visualization

html.Div(

id='map-id',

className='col s12 m6',

),

# Pie Chart

dcc.Graph(

id='breed-pie-chart',

figure={}

)

])

# Criteria for filtering based on rescue type

RESCUE\_CRITERIA = {

'Water': {

'breeds': ['Labrador Retriever Mix', 'Chesapeake Bay Retriever', 'Newfoundland'],

'gender': 'Intact Female',

'age\_range': (26, 156)

},

'Mountain': {

'breeds': ['German Shepherd', 'Alaskan Malamute', 'Old English Sheepdog', 'Siberian Husky', 'Rottweiler'],

'gender': 'Intact Male',

'age\_range': (26, 156)

},

'Disaster': {

'breeds': ['Doberman Pinscher', 'German Shepherd', 'Golden Retriever', 'Bloodhound', 'Rottweiler'],

'gender': 'Intact Male',

'age\_range': (20, 300)

}

}

@app.callback(

[Output('table', 'data'),

Output('breed-pie-chart', 'figure'),

Output('test-output', 'children')],

[Input('update-button', 'n\_clicks')],

[State('rescue-type-radio', 'value'),

State('dog-breed-dropdown', 'value')]

)

def update\_output(n\_clicks, rescue\_type, dog\_breed):

filtered\_df = df

# Filter by rescue type

if rescue\_type in RESCUE\_CRITERIA:

criteria = RESCUE\_CRITERIA[rescue\_type]

filtered\_df = df[df['breed'].isin(criteria['breeds']) &

(df['age\_upon\_outcome\_in\_weeks'] >= criteria['age\_range'][0]) &

(df['age\_upon\_outcome\_in\_weeks'] <= criteria['age\_range'][1]) &

(df['sex\_upon\_outcome'] == criteria['gender'])]

# Filter by dog breed

if dog\_breed != 'all':

filtered\_df = filtered\_df[filtered\_df['breed'] == dog\_breed]

# Create pie chart with combined small categories

pie\_data = filtered\_df['breed'].value\_counts(normalize=True).reset\_index()

pie\_data.columns = ['breed', 'percent']

threshold = 0.02

mask = pie\_data['percent'] > threshold

tail = pie\_data.loc[~mask]

others = pd.DataFrame({'breed': ['Others'], 'percent': [tail['percent'].sum()]})

pie\_data = pd.concat([pie\_data[mask], others], ignore\_index=True)

pie\_chart\_figure = px.pie(pie\_data, names='breed', values='percent', title='Distribution of Breeds')

return filtered\_df.to\_dict('records'), pie\_chart\_figure, f"Filtered by: {rescue\_type} and {dog\_breed}"

@app.callback(

Output('table', 'style\_data\_conditional'),

[Input('table', 'selected\_rows')]

)

def highlight\_selected\_row(selected\_rows):

if not selected\_rows:

return []

return [{

'if': {'row\_index': selected\_rows[0]},

'backgroundColor': 'yellow',

'color': 'black'

}]

@app.callback(

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

[Input('table', 'selected\_rows'),

Input('table', 'data')]

)

def update\_geolocation(selected\_rows, table\_data):

dff = pd.DataFrame.from\_dict(table\_data)

if not selected\_rows:

row = 0

else:

row = selected\_rows[0]

animal\_name = dff.iloc[row, 9]

animal\_breed = dff.iloc[row, 4]

return [

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

center=[30.75,-97.48], zoom=10, children=[

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

dl.Marker(position=[dff.iloc[row, 13], dff.iloc[row, 14]], children=[

dl.Tooltip(f"Breed: {animal\_breed}, Name: {animal\_name}"),

dl.Popup([

html.H1("Animal Name"),

html.P(animal\_name)

])

])

])

]

if \_\_name\_\_ == '\_\_main\_\_':

app.run\_server(debug=True)

--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

## CRUD:

from pymongo import MongoClient

from bson.objectid import ObjectId

class CRUD(object):

""" CRUD operations for MongoDB """

def \_\_init\_\_(self, USER='aacuser', PASS='aacPass', HOST='nv-desktop-services.apporto.com', PORT=32080, DB='AAC', COL='animals'):

# Initialize Connection

self.client = MongoClient('mongodb://%s:%s@%s:%d' % (USER, PASS, HOST, PORT))

self.database = self.client[DB]

self.collection = self.database[COL]

def create(self, data):

"""Insert a new document into the collection."""

if data:

try:

self.collection.insert\_one(data)

return True

except Exception as e:

print("Error occurred:", e)

return False

else:

print("Data not provided")

return False

def read(self, query):

"""Query documents from the collection."""

try:

result = list(self.collection.find(query))

return result

except Exception as e:

print("Error occurred:", e)

return []

def update(self, query, data):

"""Update document(s) in the collection."""

try:

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

return result.modified\_count

except Exception as e:

print("Error occurred:", e)

return 0

def delete(self, query):

"""Delete document(s) from the collection."""

try:

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

return result.deleted\_count

except Exception as e:

print("Error occurred:", e)

return 0