# A screenshot of a computer program AI-generated content may be incorrect.A screenshot of a computer program AI-generated content may be incorrect.A screenshot of a computer AI-generated content may be incorrect.A screenshot of a computer program AI-generated content may be incorrect.A screenshot of a computer program AI-generated content may be incorrect.A screenshot of a computer AI-generated content may be incorrect.A screenshot of a computer AI-generated content may be incorrect.A screenshot of a computer AI-generated content may be incorrect.

# Grazioso Salvare Dashboard — Project Two (CS 340)

**Author:** Jaden Mericle

## Overview and Required Functionality

This dashboard allows Grazioso Salvare to explore Austin Animal Center dog records and quickly identify candidates for different rescue scenarios. The interface includes a rescue-type selector and an age filter, an interactive data table that updates in real time, a bar chart of the most common breeds in the current view, and a geolocation map that plots dog locations. The header displays the Grazioso Salvare logo and my identifier. The screenshots included in this document demonstrate the starting state of the dashboard, the three required rescue filters (Water Rescue; Mountain or Wilderness Rescue; Disaster or Individual Tracking), and the reset state. In each case, the data table, chart, and map update consistently with the selected filters.

## Tools and Rationale

The application follows an MVC pattern. MongoDB serves as the model because the AAC records are semi-structured and benefit from a flexible schema; documents can include optional fields (such as multiple location fields), and they map directly to Python dictionaries, which makes projection and filtering straightforward. Dash provides the view and controller: components such as the dropdown, checklist, data table, and graphs define the user interface, while Dash callbacks implement the controller logic that reacts to user input, runs queries through the CRUD module, and updates the visual components. Plotly is used to render the bar chart and an OpenStreetMap-based geolocation view, which requires no additional packages and works reliably in the Jupyter environment.

## Filter Semantics

Rescue-type filtering is implemented by mapping each rescue scenario to a set of preferred breeds using case-insensitive regular expressions. The “Water Rescue” option targets breeds such as Labrador Retriever, Newfoundland, and Chesapeake Bay Retriever. “Mountain or Wilderness Rescue” targets working and herding breeds such as German Shepherd, Border Collie, and Bloodhound. “Disaster or Individual Tracking” focuses on tracking-capable and versatile working breeds, including Doberman Pinscher, German Shepherd, and Golden Retriever. The age filter constrains results to dogs two years old or younger. The code converts the string field age\_upon\_outcome into months so that “years,” “months,” “weeks,” and “days” are handled consistently and the comparison is accurate.

## How to Run and Reproduce

The dashboard runs inside JupyterLab using JupyterDash. MongoDB must be running with the AAC dataset loaded in the aac.animals collection. The CRUD module is imported as animal\_shelter and connects using the configured credentials for the aacuser account. After opening the notebook, running all cells starts the app in JupyterLab; the starting page shows the logo and identifier, the rescue-type dropdown, the age checkbox, the reset button, the interactive data table, the breed chart, and the map. Selecting any rescue type filters the collection by breed and refreshes the table, chart, and map. Toggling the age filter updates the view to include or exclude dogs older than two years. Selecting a row in the table recenters the map on that animal’s coordinates. Pressing Reset returns the dashboard to its initial state.

## Implementation Notes

To prevent crashes from ObjectId values, the projection removes the \_id field before converting records to a DataFrame. Because AAC geo fields are not uniform across all records, the code unifies coordinates by creating lat and lon columns from location\_lat/location\_long, with a fallback to outcome\_latitude/outcome\_longitude when needed. If a view has missing or nonnumeric coordinates, the map displays a stable placeholder state centered on Austin and then recenters when valid points are available. The breed chart aggregates the top fifteen breeds in the current view and rotates labels for readability. All components are wired through Dash callbacks so that the table, chart, and map stay synchronized.

## Challenges and Resolutions

The first challenge was inconsistent geographic data. Some records use location\_\* fields while others only include outcome\_\*. The solution was to compute unified lat and lon columns with a simple fallback rule and to coerce nonnumeric values safely. The second challenge was age parsing. The helper function converts the age\_upon\_outcome strings—years, months, weeks, or days—into months, enabling a single numeric comparison for the “two years or younger” requirement. A third challenge involved breed matching for mixed-breed strings; case-insensitive regex queries handle partial matches reliably and work well with hyphenated or compound breed names. Finally, the data table initially failed when \_id was present; removing it in the MongoDB projection resolved the issue cleanly.

## Resources

Dash User Guide (component structure and callbacks), Plotly Express (charts and OpenStreetMap maps), and PyMongo (queries and projections) were used to implement the dashboard and connect to MongoDB in Python.