## About the Project/Project Title

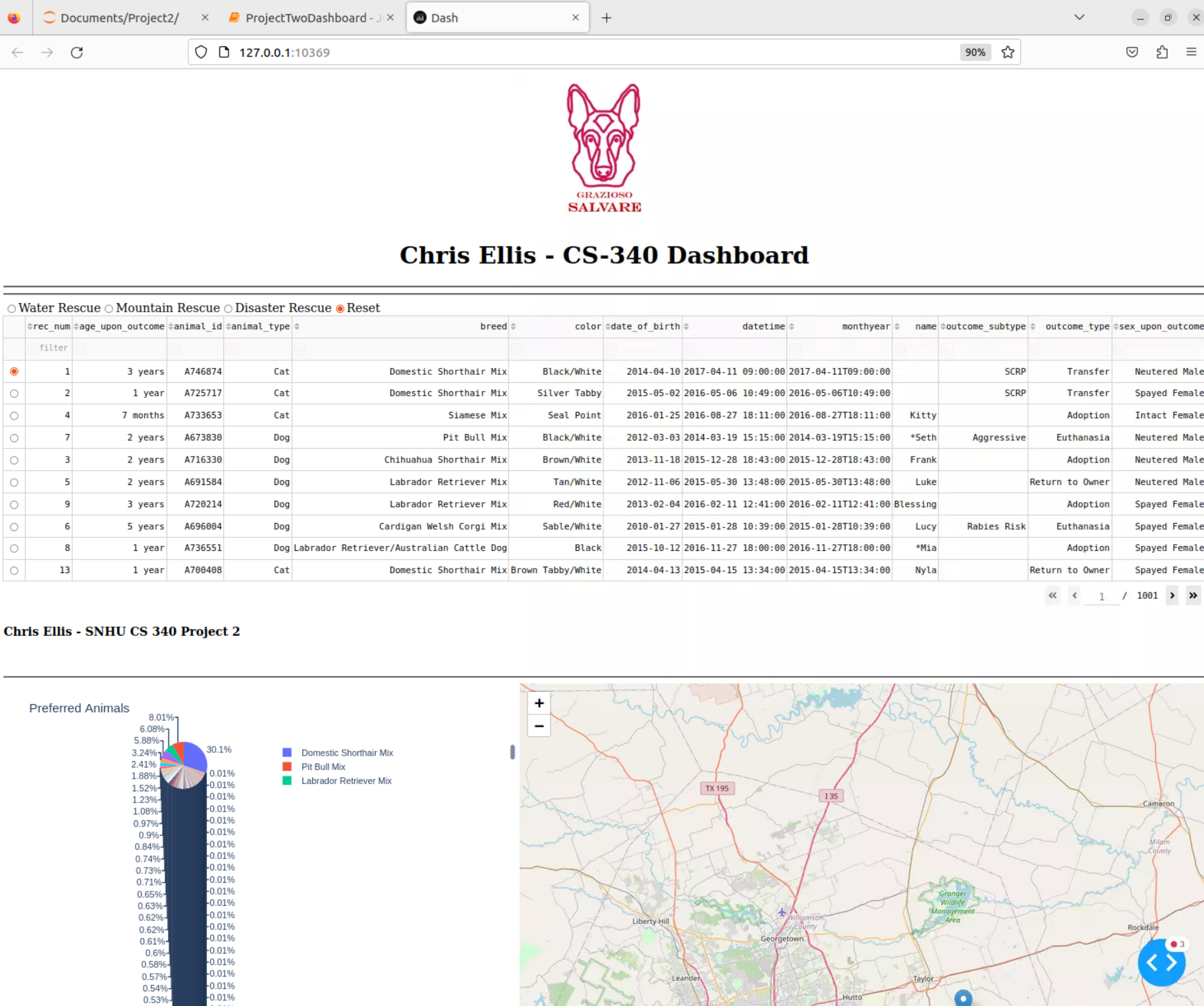
The python CRUD module provides the functionality for create/read/update/delete operations for the Austin Animal Shelter project for CS340 at SNHU. This module instantiates a connection to a MongoDB database using the pymongo library driver, and database configuration provided by a user, and allows the user to call the CRUD functions.

MongoDB is used for the database/state storage layer of the application. MongoDB is a NoSQL database that stores records as JSON ‘Documents’ and allows efficient querying around that style of record. In this application it provides storage for the data related to the animals in the AAC animal shelter.

Python is used for the backend code that provides a code-layer that can be used to interact with the AAC database in order to perform CRUD operations on the mongo database. This code can be called from different ‘frontend’ applications as it is a python library that can be imported, allowing multiple different use cases to take advantage of code re-use for accessing the database.

The test implementation is performed in a Jupyter notebook file, which uses the IPython interactive command shell to execute Python code inside of the notebook. The test notebook ProjectTwoDashboard.ipynb can interactively import and run the Python code from this project’s animalShelter.py file, allowing quick setup and re-runs of the code. The provided Jupyter notebook covers testing all of the pieces of CRUD functionality in the code including create/read/update/delete functions, and also some edge case testing around places where certain specific results are expected that do not return data.

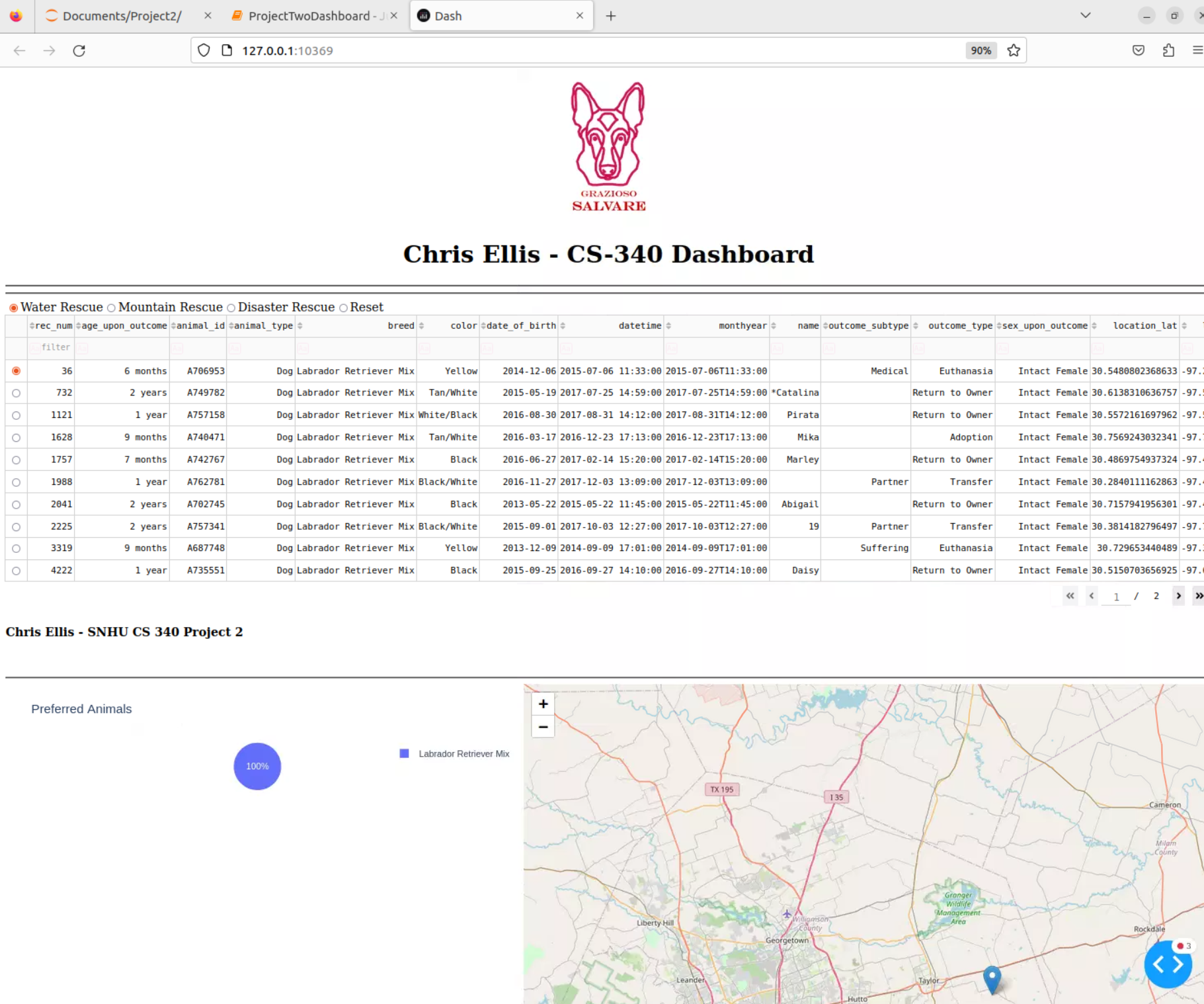
The test implementation dashboard demonstrates multiple functionalities, it’s initial state showing a data table of all records in the database (shown in the below screenshot):



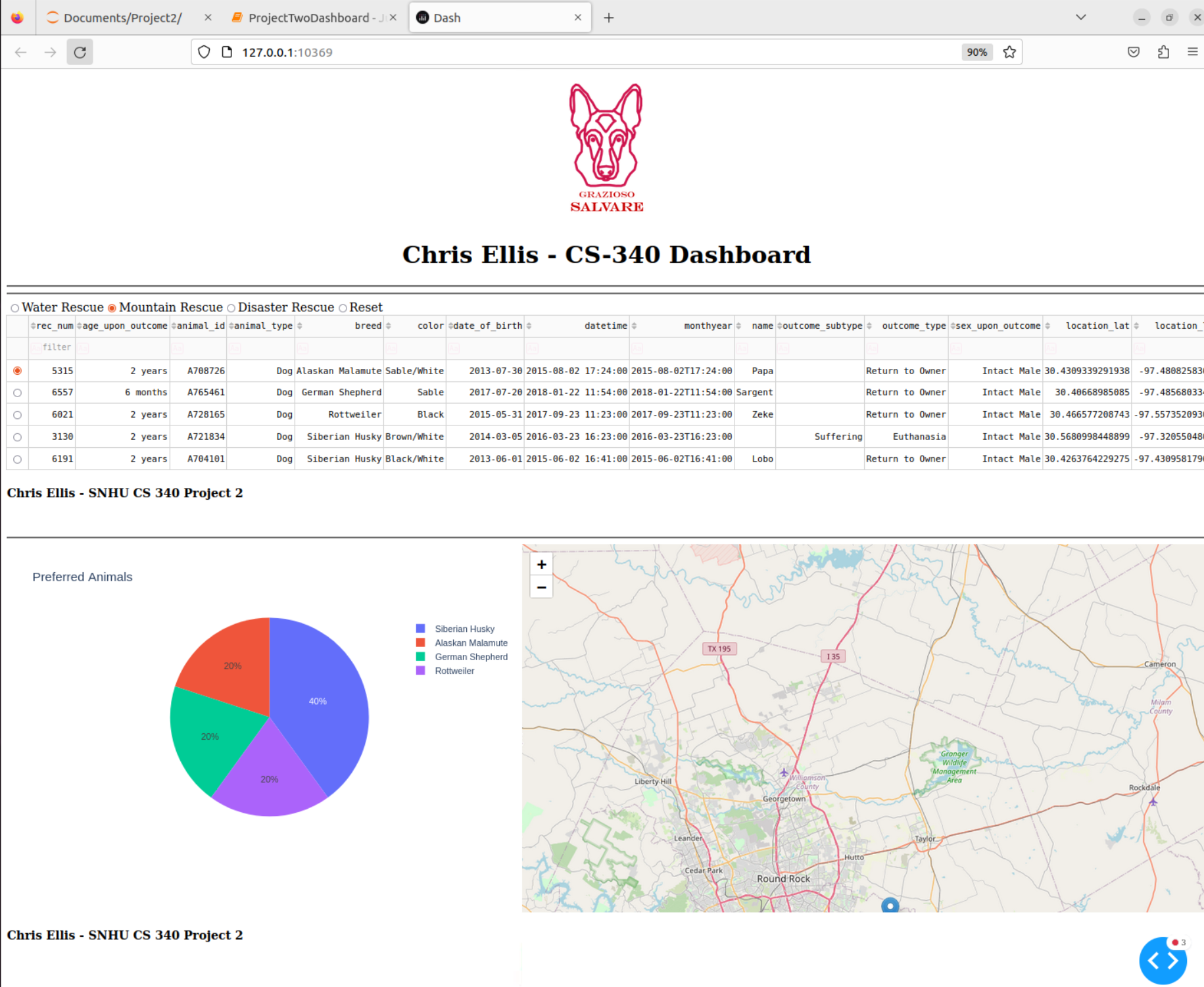
The dashboard can also filter using the specific radio buttons for different queries in the database in order to return specific sets of animals, namely ‘Water Rescue Dogs’, ‘Mountain Rescue Dogs’, and ‘Disaster Rescue Dogs’, in addition to a radio button that will reset the data table back to it’s original state.

These Filtered options are demonstrated below, as you can see, the map and pie chart diagrams will update (the map will default to the location of the first animal as it is auto selected, but any data table row can be selected to see an animals location). The pie chart will show the percentage of each breed including all of the animals found by the specific filter currently active.

Water Rescue Filter:

****

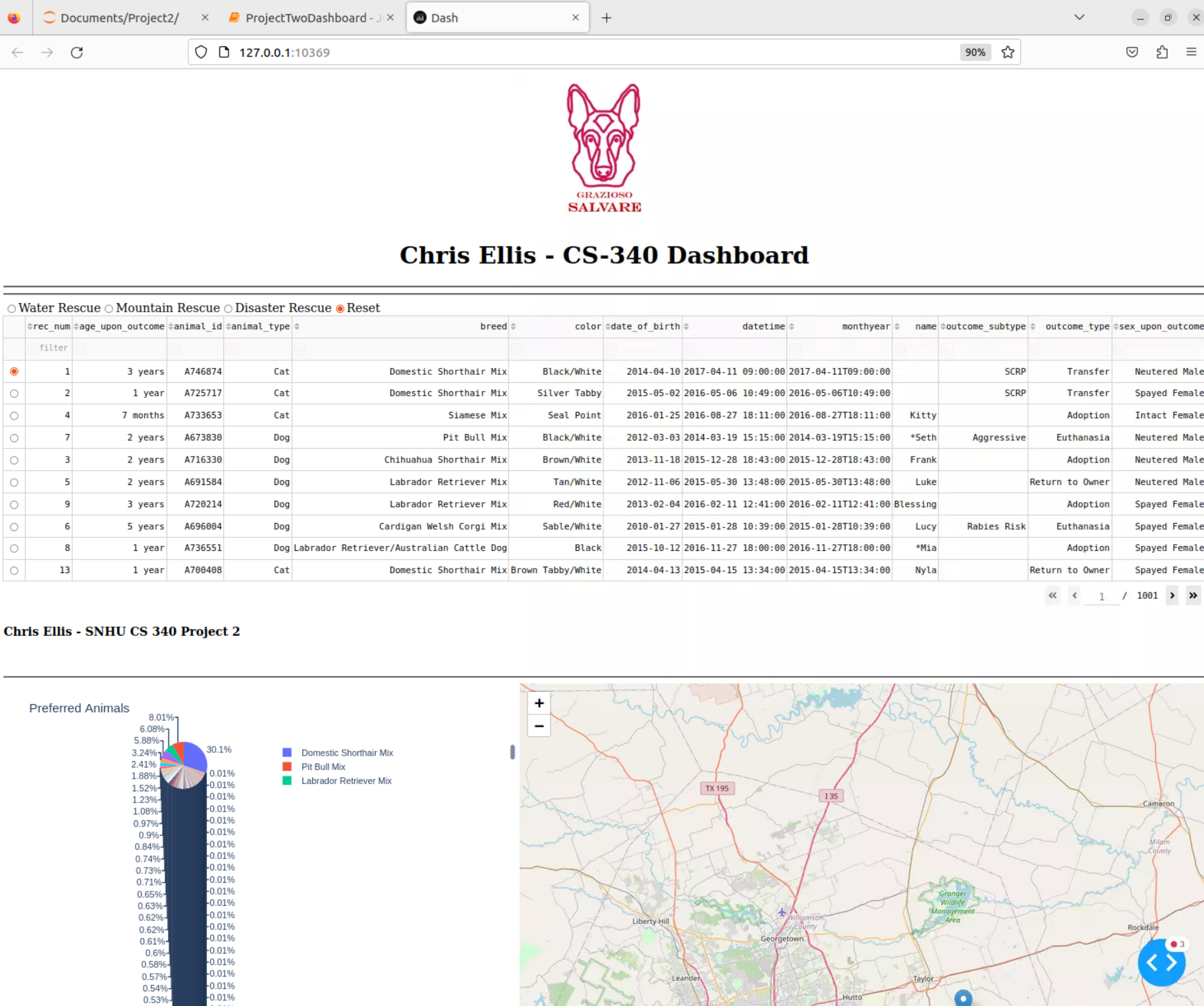
Mountain Rescue Filter:

****

Disaster Rescue Filter:



Reset Filter:

****

**Tool Choice Rationale:**

A number of tools were used in the creation of this project, including MongoDB for the database, Python for the backend code, Plotly Dash for the frontend code, and Jupyter Notebook to execute the frontend interactively.

MongoDB was chosen for the database layer, and it has a few specific advantages for developing a project like this. It is schemaless, so there is no complex model of tables to set up, just a JSON document and whatever specific information is needed in it. This also allows rapid changes to the documents like adding new fields without having to migrate a schema as well. The PyMongo library is used to call specific functions in MongoDB from python code, making the knowledge of Mongo transferable directly to writing things like queries in Python in order to access the database.

Python was used to write the backend code and frontend code for the application. Python is an interpreted language so it does not need to be compiled, meaning the frontend portion can be loaded very quickly after making changes in a program like Jupyter Notebook by clicking a button to reset the state and run the program. The backend code being Python means it can be imported into the separate frontend that runs in Jupyter notebook simply by putting the file in the same directory as the notebook and importing it.

The frontend uses the Plotly Dash library to create a dynamically updating frontend dashboard (<https://dash.plotly.com/>). As the application follows the MVC pattern (model-view-controller), with mongoDB being the model portion of that pattern, the view and controller functionality is implemented in plotly dash using callback functions for the controller (that use the underlying backend CRUD module mentioned previously to connect to the database), and internal plotly functions to return HTML for the views portion of the application, or what the user actually sees. The underlying callbacks and dynamically update the views as the user performs certain actions, as mentioned in the above sections on functionality.

**Steps Taken to Complete the Project**

This project was developed incrementally, starting with the database. MongoDB was setup by loading data from a CSV file as documents in the ‘animals’ collection in a database created on the Mongo instance called ‘AAC’. In addition a separate non-admin user was created in order to have role based access control so the application would not need to use the administrative user to query data. In addition to this, test queries were performed to ensure access to the specific data and fields that would be needed later in the project.

The second portion completed was the backend CRUD module animalShelter.py, which is python code used to query data from MongoDB and turn that data into objects accessible in Python. This code uses the Pymongo library in order to call MongoDB functions and performs error checking to ensure correct data is returned. It implements all of the CRUD functionality including Creating new records, Reading/Querying existing records, Updating one or multiple fields on a number of existing records, and deleting records. In addition the functions can be provided queries in order to limit what that function will apply to, such as deleting all records of animal type ‘Dog’, or updating all ‘Labrador’ breed animals to have some other specific fields.

Initially the backend code was tested by building Jupyter Notebook that implemented tests for each of the CRUD functions, to ensure they were working properly and the import of the backend code was functioning correctly, along with it being able to connect to the database with user provided credentials given in Jupyter and passed to the backend code and subsequently MongoDB.

After this initial testing, a basic frontend was made in Jupyter Notebook using the plotly dash library, which displayed a data table and a geo-map that would show the latitude/longitude location of the animal currently selected in the data table. The initial data table only presented a query of all of the animals in the database, but was made more user friendly by implementing pagination, and functionality to sort the existing records.

The final version of the dashboard added radio buttons to the data table that run pre-defined filters for different types of rescue animals, filtering the table down to just the animals matching those specific rescue types. In addition a pie chart was added that will dynamically update a percentage-based chart of the breed of each of the animals returned by the currently selected radio button query.

The map will dynamically update to show the location of the selected animal, or upon changing of the query by selecting any radio button, initialize itself to the first animal returned by that query.

**Challenges**

There were a few challenges encountered when creating this project. Most of the challenges were due to debugging errors in the libraries used, as the developers were not very familiar with all of them. Plotly dash specifically has a large number of components and the errors returned on the dashboard do not contain line numbers from the traceback, so one may have to dig through many components that use the same naming scheme in order to determine which of the components is throwing the error. This was accomplished by commenting-out or disabling the callback functions and enabling one at a time in order to narrow down the location of the component that was throwing the error.

Another area that included challenges was any place that different layers of the framework interact, such as connecting python to MongoDB, or importing and testing backend code in the Jupyter Notebook, to debugging issues on the dashboard as they may be due to errors in either the frontend, backend, or MongoDB. This was a great learning experience in narrowing down cryptic error information to a specific piece of the technology stack that was causing an issue.

**Motivation**

CS340 is about full stack client-server development, and as such involves creating frontend and backend code, and storing data for the applications in a database. Developing a library to handle CRUD operations to the database allows a large amount of code-reuse and standardization for those operations which are extremely commonplace in a web app that stores state in a database.

**Reproducing The Project:**

## Getting Started

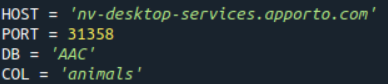
To get started using the CRUD library, you must first import the module in your python code:



You will then need to instantiate an object to call the operations on the module class, providing your database username and password, additional configuration of the server details can currently only be handled inside the module’s hardcoded values:



Server configuration:

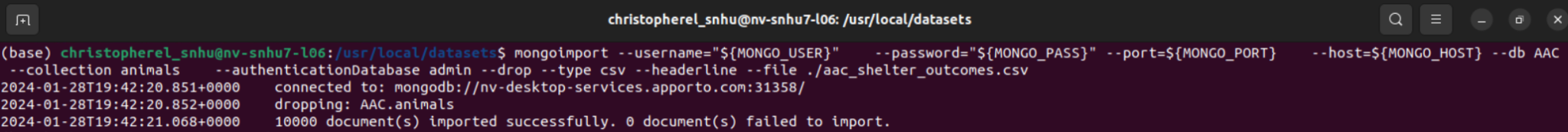


## Installation

This module requires manually copying the animalShelter.py file to a location where it can be imported into a python environment (required python version >3.9). To install the dependencies you must also run:

pip install pymongo

Use of this module also requires a running MongoDB database which is not covered in this installation guide. Populating the instance with the AAC database can be done by importing the aac database from the provided .csv file as shown:

****

The provided Jupyter notebook file Script.ipynb can be opened directly in Jupyter notebook, but must be present in the same directory as the animalShelter.py in order to install the

## Usage

*Use this space to show useful examples of how your project works and how it can be used. Be sure to include examples of your code, tests, and screenshots.*

### Code Example

Import the library, initialize connection to MongoDB and create a AnimalShelter object to perform CRUD operations

| from animalShelter import AnimalShelter  *# create an AnimalShelter object*  CRUD = AnimalShelter('aacuser', 'aacuserpassword1234') |
| --- |

**The create method can be used to add new records to the database, providing a data dictionary of key-value pairs used in the animal document:**

| *# Create a new record in the database*  CRUD.create({  "age\_upon\_outcome": "1 year",  "animal\_id": "test\_id",  "animal\_type": "Dog",  "breed": "Domestic Shorthair Mix",  "color": "White",  "date\_of\_birth": "2014-04-10",  "datetime": "2017-04-11 09:00:00",  "monthyear": "2017-04-11T09:00:00",  "name": "Test Animal",  "outcome\_subtype": "SCRP",  "outcome\_type": "Transfer",  "sex\_upon\_outcome": "Neutered Male",  "location\_lat": 30.5066578739455,  "location\_long": -97.3408780722188,  "age\_upon\_outcome\_in\_weeks": 156.767857142857  }  ) |
| --- |

The read function can be used by providing query key-value pairs to a dictionary, and it will return a list of result documents from the database. NOTE: You must iterate over the returned list to view all of the results:

| *# Test read function with created test animal*  query\_result = CRUD.read({"name": "Test Animal"})  *# print read function results*  print()  print("Results from correct read query:")  *# loop through all query results returned and print individually*  for document in query\_result:  print(document) |
| --- |

The update function can be used to update fields on any documents that match the provided query, NOTE: Both query and update\_data parameters must be non-empty or the function will not perform an update. In this example any records that match the “name” field “Test Animal” will have their “color” field changed to “Red” or if that field doesn’t exist, it will be added with the value red.

| # call update  CRUD.update({"name": "Test Animal"},{"color": "Red"}) |
| --- |

In order to call the delete function, you must provide a similar query key/value pair as the read function, the delete function will delete any records matching this, and return the number of records deleted:

| CRUD.delete({"name": "Test Animal 2"}) |
| --- |

### Tests

The tests can be run be loading the Script.ipynb notebook file into a jupyter notebook instance, and clicking the run button

Test code examples:

Create:

*# Create a new record in the database*

print(CRUD.create({

"age\_upon\_outcome": "1 year",

"animal\_id": "test\_id",

"animal\_type": "Dog",

"breed": "Domestic Shorthair Mix",

"color": "White",

"date\_of\_birth": "2014-04-10",

"datetime": "2017-04-11 09:00:00",

"monthyear": "2017-04-11T09:00:00",

"name": "Test Animal",

"outcome\_subtype": "SCRP",

"outcome\_type": "Transfer",

"sex\_upon\_outcome": "Neutered Male",

"location\_lat": 30.5066578739455,

"location\_long": -97.3408780722188,

"age\_upon\_outcome\_in\_weeks": 156.767857142857

}

))

Read:

*# Test read function with created test animal*

query\_result = CRUD.read({"name": "Test Animal"})

*# print read function results*

print()

print("Results from correct read query:")

*# loop through all query results returned and print individually*

for document in query\_result:

print(document)

**Update:**

| *# call update*  CRUD.update({"name": "Test Animal"},{"color": "Red"})  print()  print()  *# show the updated field for Test Animal has changed from Black to Red*  print("Test Animal color field should return value with color field 'Red', actual query result: ")  update\_result = CRUD.read({"name": "Test Animal"})  for document in update\_result:  print(document)  print(15 \* '-')  print() |
| --- |

### Delete:

| *# creating 3 objects to delete*  print("Testing Delete functionality...")  print("Creating 3 'Test Animal 2' objects...")  CRUD.create({"name": "Test Animal 2"})  CRUD.create({"name": "Test Animal 2"})  CRUD.create({"name": "Test Animal 2"})  deleted = CRUD.delete({"name": "Test Animal 2"})  *# call delete on Test2 Animal objects*  print(f"Deleted: {deleted} objects")  *# confirm delete by checking for Test2 Animal records*  deleted\_read = CRUD.read({"name": "Test Animal 2"})  print(f"This should be an empty list with no records after deletion: {deleted\_read}")  print(15 \* '-')  print()  *# delete original create object to cleanup test*  CRUD.delete({"name": "Test Animal"}) |
| --- |

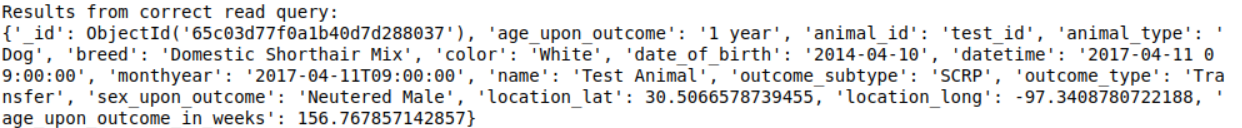
|  |
| --- |

### Screenshots

Create test returns true by printing the successful connection string to the DB, and printing the create method call, which will return true (second line shown) on a successful data insertion:



The read test will return the sample object inserted, in this case an animal named “Test Animal” with it’s specific provided data, the test query uses the name “Test Animal” as the key-value pair for lookup.



Running the test suite should produce the following results for each function:

****

## Roadmap/Features (Optional)

This project is feature complete at this point, but can include optional roadmap items like making better use of the constructor to handle all database connection parameters, or improvements like a suite of unit tests.

## Contact

Your name: Christopher Ellis christopher.ellis12@snhu.edu