Project Two README

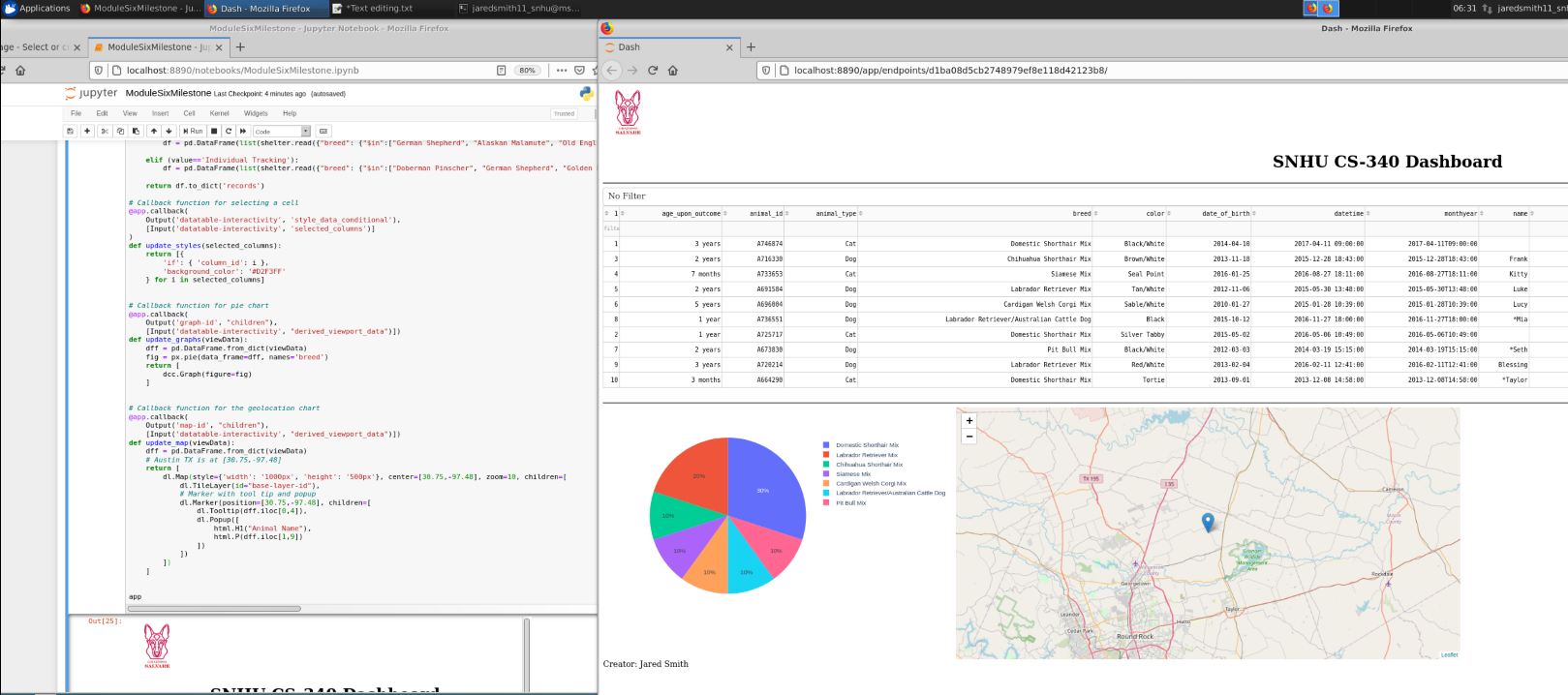
CS-340 23EW4

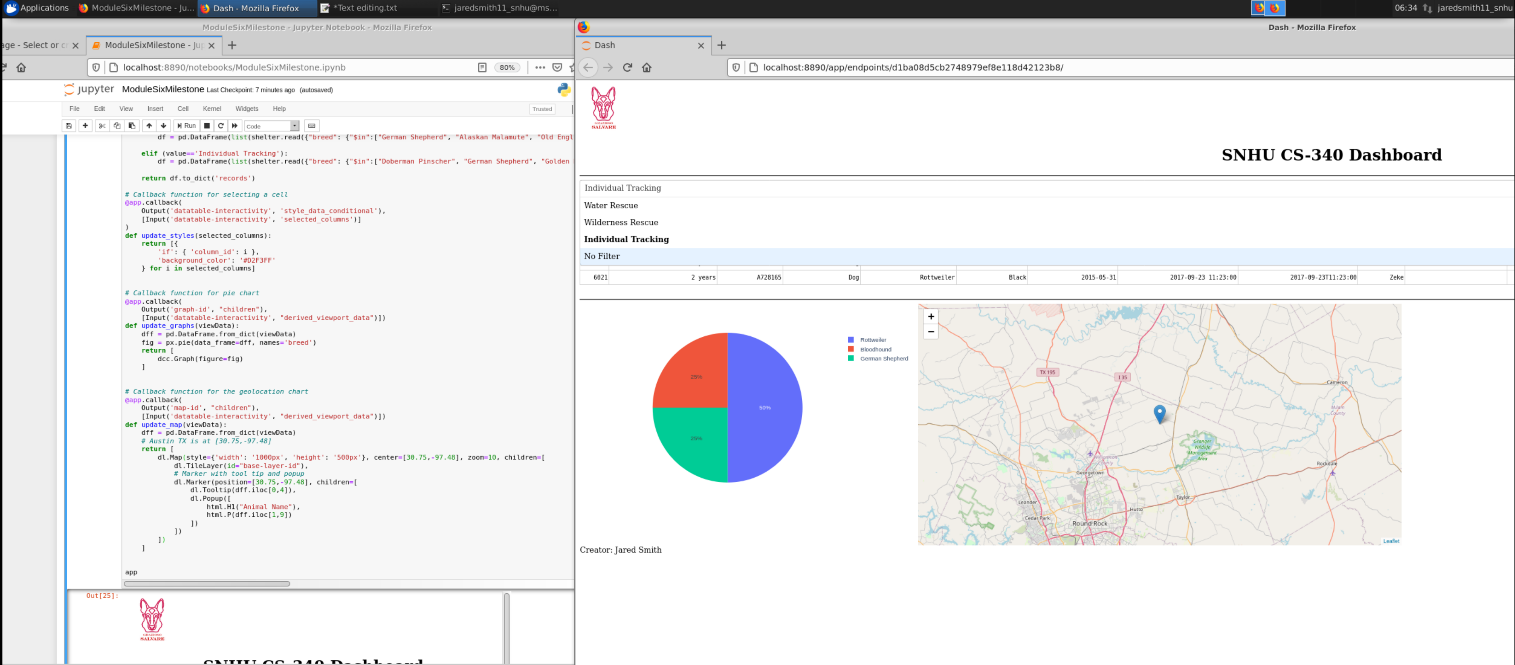
Jared Smith

4/13/23

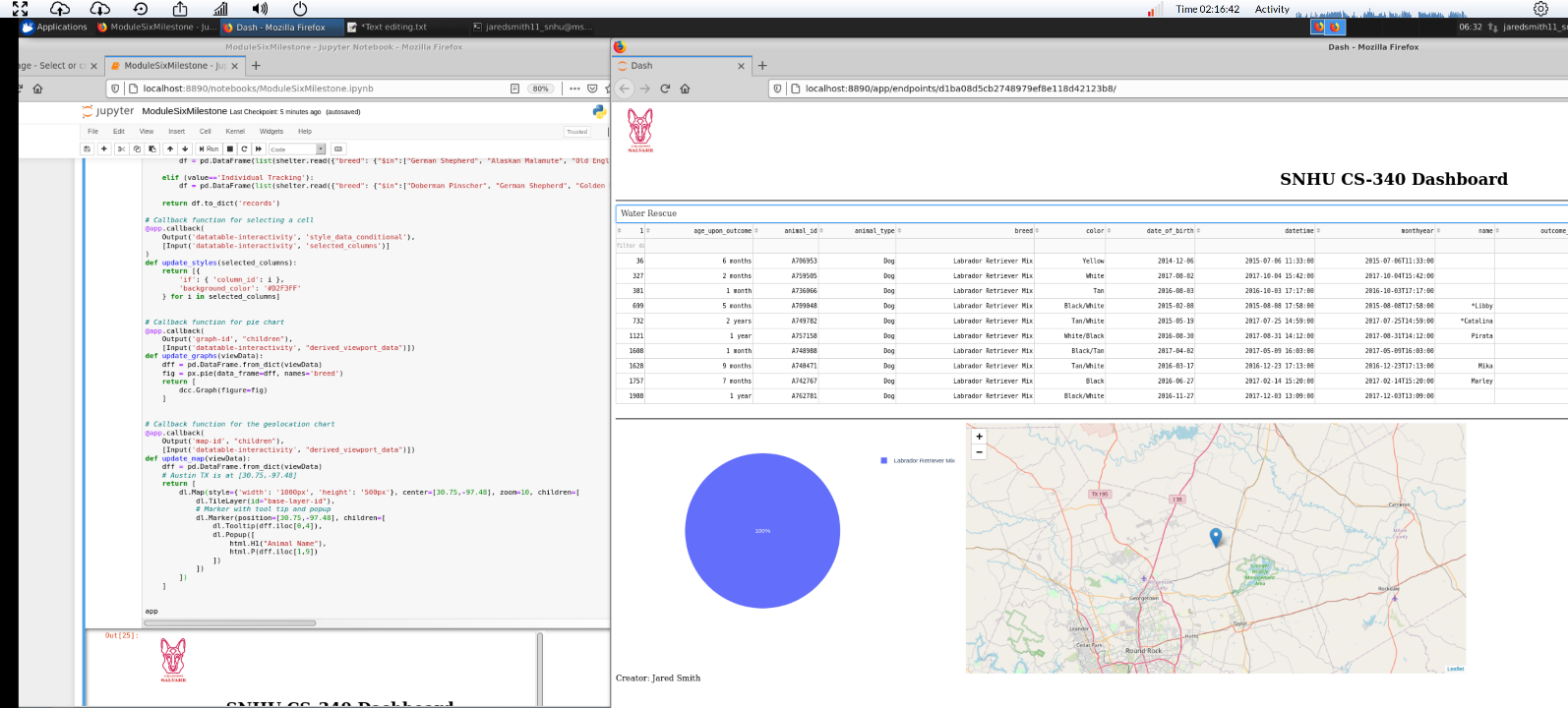
**FUNCTIONALITY**

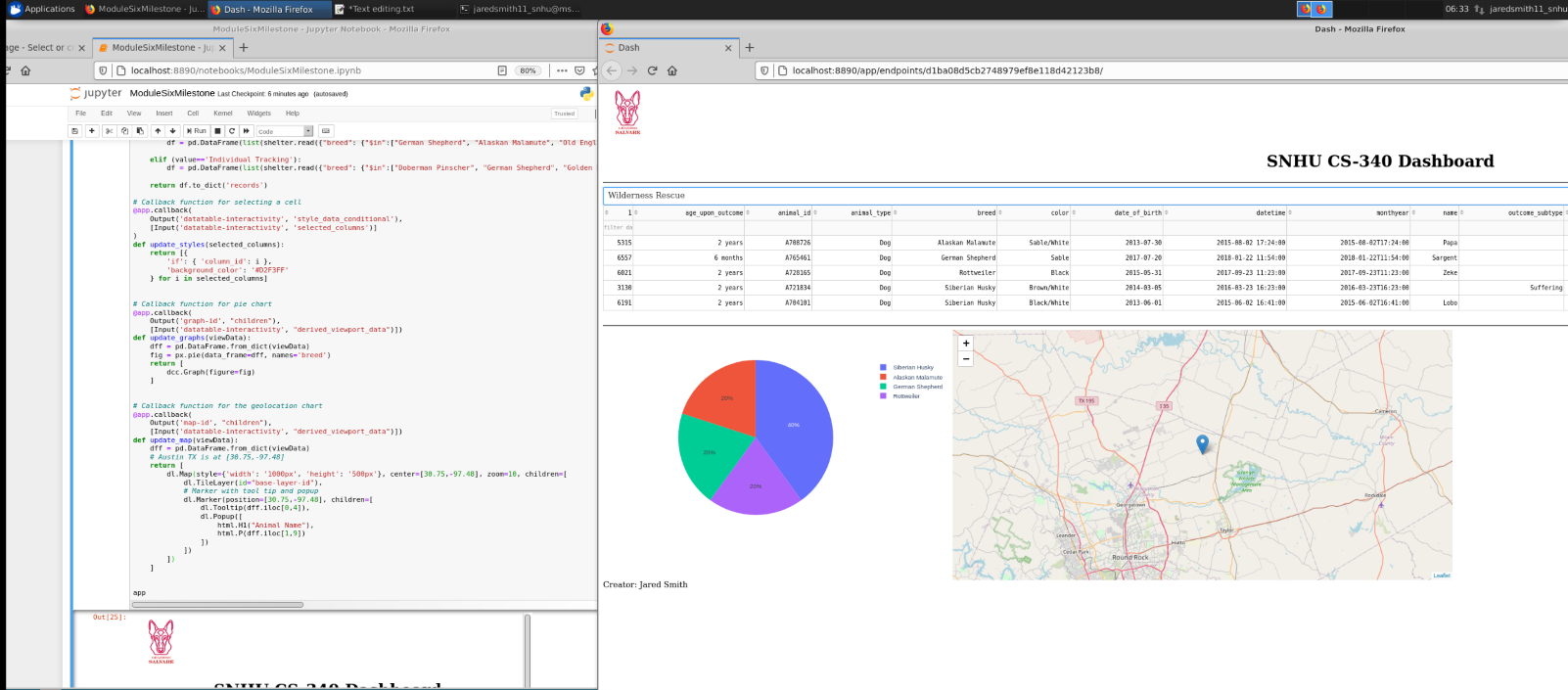
The purpose of this application is to provide a notebook in which the user can display a table of rescue animals from the AAC shelter. The app also displays the geographic location of the selected entry from the table and a chart that describes the breed breakdown of the data currently on the table. The table also comes with several prebuilt filtering options for common rescue operations, and the table and charts update appropriately with the filtering.

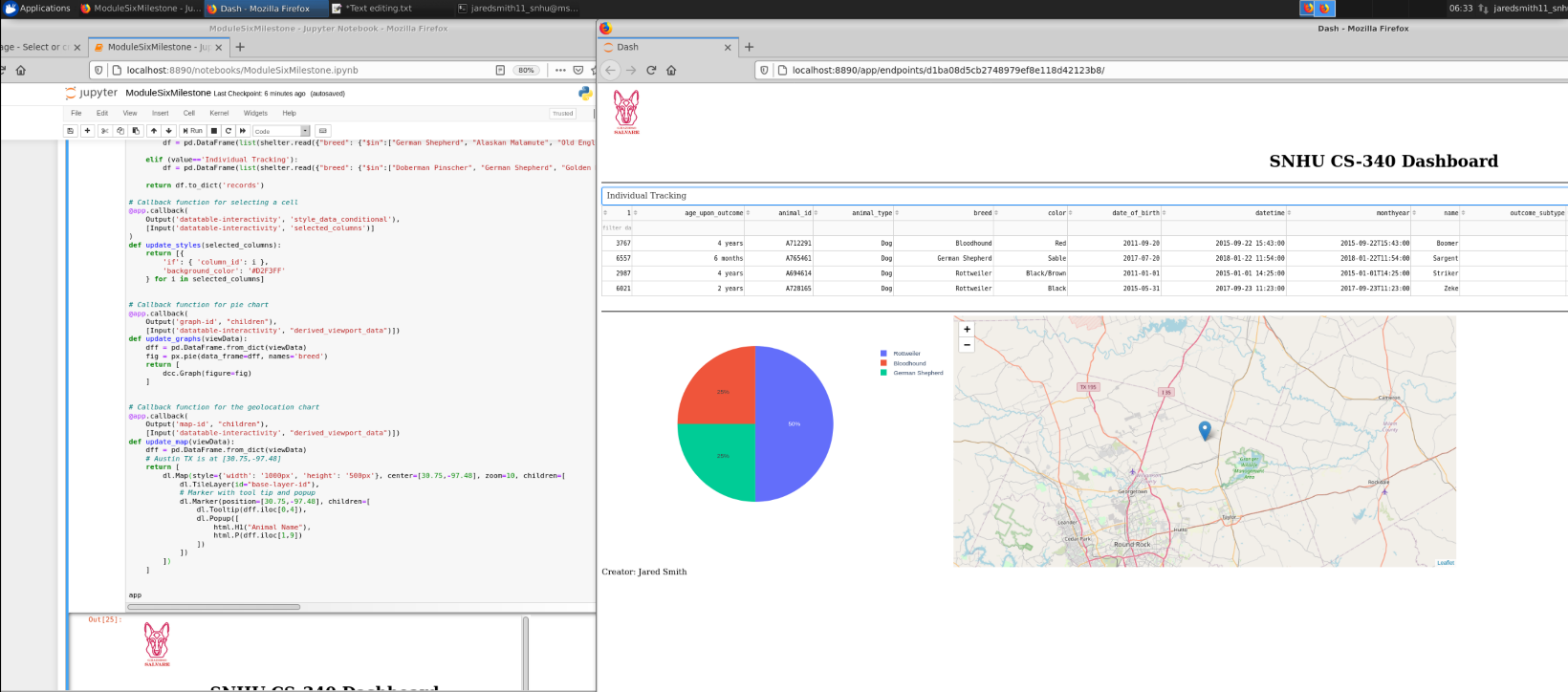
The app’s starting state displays the unfiltered table, and by default shows the location of the first entry on the table. 

The default filtering options filter the table down to shelter entries appropriate for water rescue, wilderness rescue and individual tracking operations for AAC. The dropdown also includes an option for no filter, which returns the table to its original, unfiltered state. 

Below are screencaps of each filter option, and the resulting data table and charts.







**TOOLS**

**MONGODB**

The base framework for this application is the NoSQL database framework MongoDB. This database framework was chosen for a few reasons. MongoDB is simple to set up, and because we’re using this on a private host, no complex configuration will be required for our use case. Secondly MongoDB has several tools for easily interacting with Python code, which will help use greatly in creating the glue that will hold our complete application together.

**PYMONGO**

That glue was created using PyMongo, a library of tools for interacting easily with our Mongo database via Python code. We used this tool to create the AnimalShelter CRUD module, which creates a connection instance to our database based on the credentials passed to it on instantiation, and provides the user access to the basic CRUD options from the Python interface, allowing for easily repeatable queries and editing of the database.

**JUPYTER NOTEBOOK**  
The interface we used for Python is Jupyter Notebook, a notebook tool that allows for quick access to repeatable coding solutions that can be used by non-programmers to access information easily.

**DASH**

Lastly, we used Dash framework and the JupyterDash module to create an HTML user interface that accesses our PyMongo module. Dash allows us to easily create an HTML interface that accepts very simple user input to query the database for prebuilt filters and display that information in a consumable way for our users.

Each of these tools can be downloaded or installed based on your operating system requirements, as detailed in each’s API.

MongoDB -

[Install MongoDB — MongoDB Manual](https://www.mongodb.com/docs/manual/installation/)

PyMongo -

[Installing / Upgrading — PyMongo 4.3.3 documentation](https://pymongo.readthedocs.io/en/stable/installation.html)

Jupyter Notebook –

[Project Jupyter | Installing Jupyter](https://jupyter.org/install)

Python –

[Download Python | Python.org](https://www.python.org/downloads/)

Dash –

[Using Dash in Jupyter and Workspaces | Dash for Python Documentation | Plotly](https://dash.plotly.com/installation)

JupyterDash –

[Using Dash in Jupyter and Workspaces | Dash for Python Documentation | Plotly](https://dash.plotly.com/workspaces/using-dash-in-jupyter-and-workspaces)

**DEVELOPMENT STEPS**

**SETTING UP THE MONGODB**

This project was developed using the Linux terminal, so all instructions and commands will reflect Linux terminal commands. The first step to getting this project running is setting up the MongoDB. Firstly, the database will needed to be started up, so we’ll instruct the terminal to find and run our Mongod\_ctl file, with the -start -noauth keywords. For me, mongo was installed under /usr/local/bin/, so my instruction looked like this:

/usr/local/bin/mongod\_ctl start -noauth

This will boot the MongoDB on local host, allowing you to set up the database and collection described in this project. Be sure to collect the localhost port information when Mongo boots, as this will be important in several following steps.

Next, we’ll use the linux terminal to import a new database and collection using Mongo’s import tool to import a csv. In the linux terminal we’ll type the following command:

mongoimport --host=localhost:54064 --db=AAC --collections=AAC --type=csv --headerline --file=/usr/local/datasets/aac\_shelter\_outcomes.csv

You will need to change the =localhost: to be followed by the port your server is hosted on, and the --file= to represent the file path your csv is located at. To import datasets from other kinds of files, check MongoDB’s API for [mongoimport](https://www.mongodb.com/docs/database-tools/mongoimport/), as there are many options. You can verify that your import was successful by entering the mongo shell by typing “mongo” into the linux terminal, and running the “show dbs” command, which should produce a result something like this:

Shape

Description automatically generated with medium confidence

If the database appears empty, try exiting the mongo shell by typing exit or pressing control+c, and giving the database a few more minutes to process your dataset.

Next, we’ll want to create a user on the database that we can use later when we re-enable authentication. After entering the mongo shell, we’ll want to type “use AAC”, to enter the database we created in the previous step. From there, we can add users using the command in the following image:

Text

Description automatically generated

You can create any number of users on this database, but for the purpose of this tutorial you will want to create at least one user with the role: “readWrite”, so we can test our CRUD functionality later. For more information on users and roles check the MongoDB [API](https://www.mongodb.com/docs/manual/tutorial/manage-users-and-roles/).

Now that we have a user on the database, we can enable authentication so only privileged users can access the database, and users can only access database functionality appropriate for their role. This can be accomplished by stopping the database with the following command

/usr/local/bin/mongod\_ctl stop

And restarting it without the -noauth tag

/usr/local/bin/mongod\_ctl start

Now, when accessing the mongo shell we will need to instruct the shell where to look for our authentication. We can do this by entering the mongo shell with the following command

mongo -authenticationDatabase “AAC” -u “aacuser” -p

By not defining the password while accessing the shell, the shell will prompt us for the password which will be hidden when entered. Access will also now be restricted to only the dbs available to the user.

Text

Description automatically generated

**PYMONGO CRUD**

Now that we have our mongo server set up with a database and at least one user with readWrite access, we can start testing our Python CRUD module. The module provided with this application has very simple CRUD operations, defined using Python scripting language and the PyMongo library for accessing the database. All the following examples were coded and executed in Jupyter Notebook, in individual in statements for ease of identification for this tutorial.

**CONSTRUCTOR**

Text

Description automatically generated

The constructor, shown above accepts the username and password as string parameters, which it passes to the MongoClient() function contained in the PyMongo library. Be sure when you import this module that you update the localhost: port with the port your server is running on, or PyMongo will not find your server.

**CREATE**

Text

Description automatically generated

The create method, shown above accepts a python dictionary as a parameter and returns a boolean indicating whether the insert method executed successfully. The dictionary is expected to be formatted like a JSON file, so it can be passed to Mongodb’s insert() method. Below is a very simple example, but more complicated dictionaries can be passed to the create method. The insert method does support nested dictionaries, and can accept a list of dictionaries to add multiple items in one function call.

Text

Description automatically generated

**READ**

Text

Description automatically generated

The read method accepts a python dictionary as a parameter, and passes it to the PyMongo .find() function, which executes a MongoDB find() function with the dictionary converted to a query. This method returns a Python cursor object which can be iterated through to retrieve a list of returned documents. Below is an example of how you might iterate through a cursor object.

Text

Description automatically generated

**UPDATE**

Text

Description automatically generated

The update method works similarly to the read method, accepting a dictionary as the key to return a query, except it also accepts a second dictionary as an update parameter. PyMongo gets a query of all documents that match the object, and updates the members specified in the update parameter for each object. This method returns an UpdatedObject object from MongoDB which has several parameters that can be accessed.

Text

Description automatically generated

**DELETE**

Text

Description automatically generated

Similar to our read method, the delete method accepts a dictionary that will act as a query on the Mongo database. The difference is instead of returning a cursor, the method drops any matching document from the query and returns a deletedObjects object from MongoDB.

Graphical user interface

Description automatically generated

**DASH FRAMEWORK**

The last component of creating this project was to implement dash framework to create a simple user interface which displays database information in a consumable way. Dash framework can be simply broken down into two parts, the html lite layout section and the python driven callback section. The app.layout section describes the visual components of the user interface. For example, the dropdown box of selectable filtering options is represented in the app.layout sections as the dash core component (dcc) object Dropdown (shown below).

Text

Description automatically generated

Each “option” in this layout statement represents an entry in the dropdown, requiring both a label, which will be the text displayed in the dropdown, and a value which will be used to determine which dropdown option is currently active. To add or update filtering options, the first step would be to edit this layout element with the updated or new dictionary items. As a note, the value= statement after the list of dropdown options sets the default option that will be active when the notebook is first started. Additional information on dash core components and html options can be found in the open source component libraries in the Dash [API site](https://dash.plotly.com/dash-core-components).

Any functionality in Dash framework is defined in “callback function”. These functions are triggered when an element designated as input is interacted with or changed, and the following python function is executed, with the return item being put to the output statement’s object and element. For example, the following is the callback function for the dropdown element. Defined in the input is (‘filter-choice’, ‘value’). These parameters describe the id of the element input will be taken from and the member or “child” of that element that will pass its value as input to the function. In this case, we are using the ‘filter-choice’ id, which describes our dcc.Dropdown element, and the ‘value’ child of that element as the variable that will be passed into the function. Below that is the Python function that will execute, which updates the dataframe for the rest of the application with a query based on the filtering option selected by the user.

Text

Description automatically generated

If, for example you wanted to update the dropdown list with additional filtering options, after you updated the dcc.Dropdown element you would need to come to this callback function, and define new functionality that would return a new filtered data frame to the data table based on the value you defined for that fitler option. For more information on callbacks, see the Dash Fundamentals, part 2. Basic Callbacks and Dash callbacks section on the [Dash API site.](https://dash.plotly.com/basic-callbacks)

**CHALLENGES AND LESSONS LEARNED**

Developing this project was very much a balancing plates experience for me. Before this term I had very little experience with Linux, SQL, and Jupyter Notebook, so learning to use tools and libraries for each was a bit of a upward climb. I found using Linux very intuitive and interesting, and learning the basics of SQL based wrappers and how to interact with NoSQL databases very valuable. Jupyter Notebook was more of a challenge, as I approached it with the very wrong assumption that it was just a Python IDE. This created a lot of complications when trying to learn to use Dash and PyMongo, and led to me spending a lot more time troubleshooting that I probably needed to. In the end, I spent some time familiarizing myself with Jupyter Notebook, and that solved most of my PyMongo problems. For anyone replicating this project, I highly recommend having a firm understanding of Python and at least some familiarity with Jupyter Notebook before starting, as trying to build an understanding of JupyterDash and PyMongo will be a bit difficult without that baseline understanding.

I also ran into a lot of problems implementing JupyterDash, as the version of Dash we were using is very out of date. Most of the very basic examples from the Dash user guide did not run in my system without some modification. What made this particularly challenging was I couldn’t get logging up and running at all throughout the project, so when there was an issue with JupyterDash or with any of my Python code, Jupyter Notebook produced no output and I had no means of obtaining error information. This means I was troubleshooting blind, and because my Dash framework was so out of date, there were no formatting solutions in the Dash user guide. The biggest example of this was our Input() for callback functions needed to be passed as python lists, or it wouldn’t return anything to the layout section and would result in no output. But that hasn’t been a requirement for several iterations of Dash framework, so none of the examples in the user guide pass single inputs to callback functions in list format.

For anyone replicating this project, I would highly recommend using the most recent version of Dash and MongoDB, as they contain a lot more elegant solutions than were implemented in this project, which was constrained to older versions of the frameworks.