Team B - Kate Spitzer, Sanabu Washizuka, Jonathan Hicks

**Coastline CRUD**

Analyzing Beach Info and Water Quality Data

horizontal line

# 

# Introduction

One of the main attractions of California is its coastline, and the vast amount of beaches that lie up and down the state. The goal of this project was to pin-point all of the beaches as well as some underlying information about each beach, such as water quality, amenities, and pet policies to name a few. This project is intended for people who would benefit from a more user-friendly way of discovering beaches that fit their wants and needs, while also providing an interactive set of visualizations about the water quality data for each beach in California. Water quality is based on the state's water quality standards, ranging from A-F meaning the beaches with higher letter grades have better water quality and lower risks of people getting sick from water contact.

## Tools

* Python/Pandas
* HTML/CSS
* PostgreSQL
* Javascript
* Requests
* Splinter
* BeautifulSoup
* Leaflet
* AnyChart
* Flask
* Heroku
* Leaflet-timeline
* ExtraMarkers

## Methodology

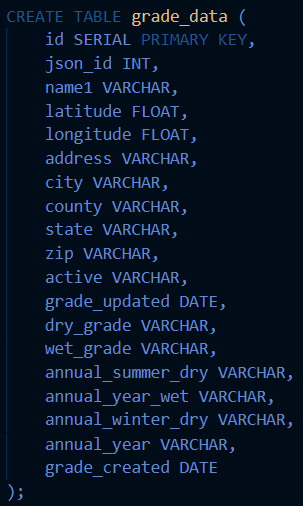
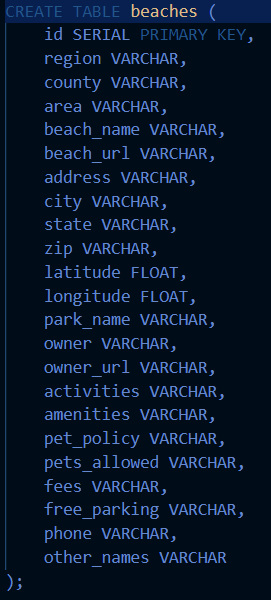
### Data Exploration

The data exploration process consisted of browsing [www.californiabeaches.com/beaches/](http://www.californiabeaches.com/beaches/) and figuring out what kind of data would be available in order to conduct an analysis and visualizations which align with the project goal. By inspecting the HTML of this webpage, it was determined that all of the data needed to create an interactive map for beaches in California was provided. The next site found during this data exploration process was <https://beachreportcard.org/> which is where the historical water quality data was discovered for the beaches.

### PostgreSQL and Data Scraping

Once it was decided which features were going to be scraped, a PostgreSQL schema was created in order to match the columns and value types of the data that is being extracted before being cleaned and transformed.

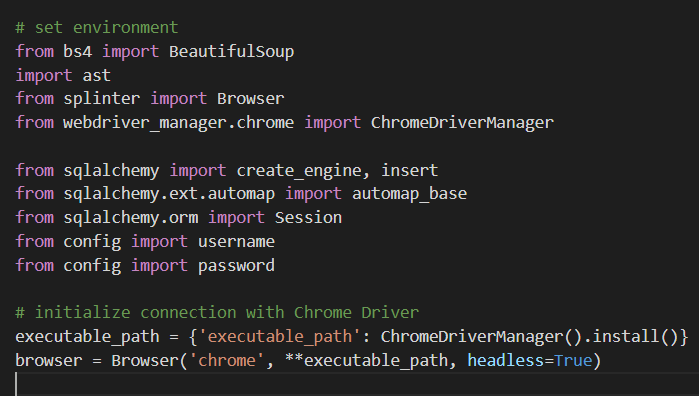
This is the schema for the postgreSQL databases:



For development purposes, Jupyter Notebook and Pandas dataframes were used to scrape and extract our data.

Beach Attribute Data

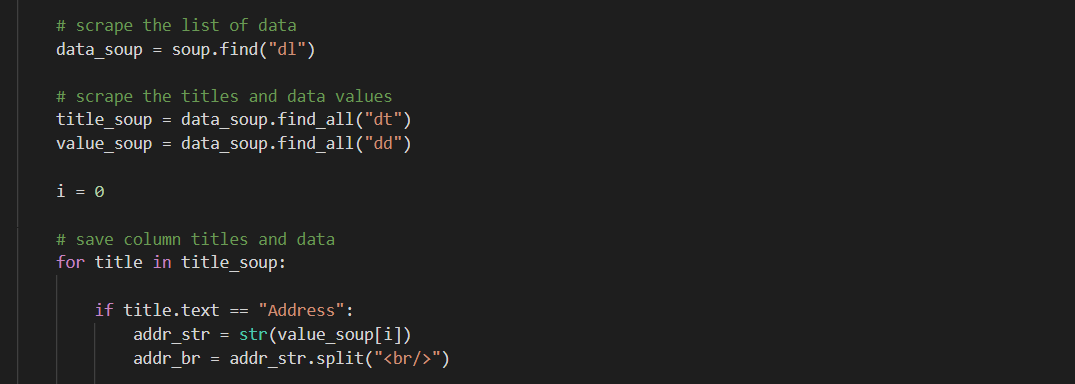
The [California Beaches](https://www.californiabeaches.com/) site was scraped using Splinter/ChromeDriver and BeautifulSoup, and the PostgreSQL database loaded using SQLalchemy. Chromedriver is initialized using the headless option, so that the browser iterations are not visible to the user.



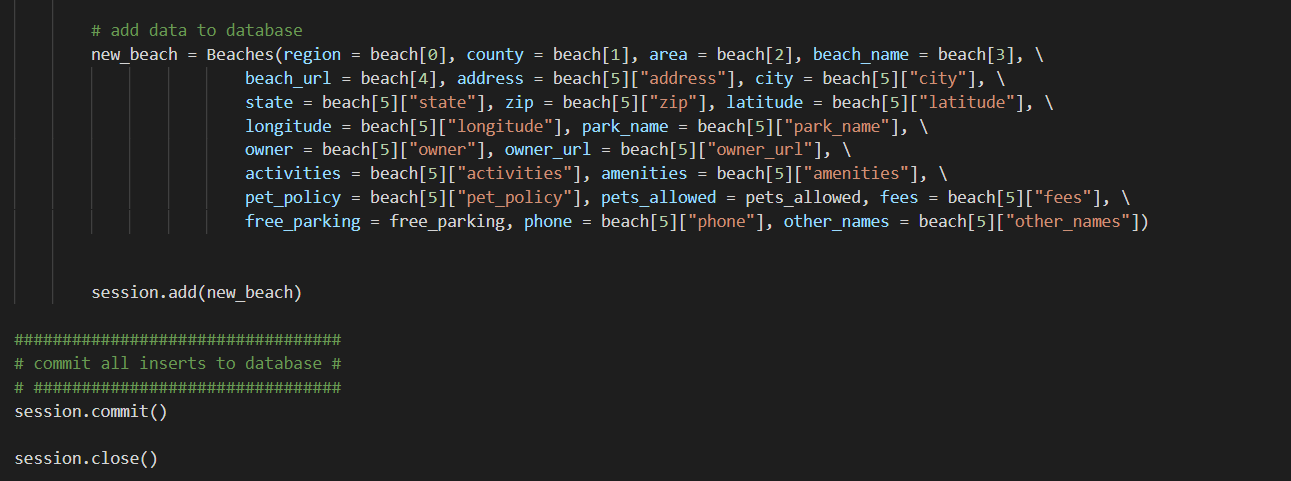
The top level Beaches page was scraped for by region and county in order to build a list of URLs pointing to each county beach page.



The page associated with each county was scraped for URLs with information on each area of the county. Again, each area URL was visited and the page scraped for URLs associated with each beach. Finally, we had a list of individual California beach URLS which would be visited and scraped for attribute data. Attribute data was located between descriptive list tags (<dl>).



Data fields were cleaned and transformed before being loaded into dataframes and saved as CSV files in development, and ultimately, loaded into our PostgreSQL database in production (scrape\_CAbeaches.py).



Water Quality Data

Historical water quality grade data (get\_histData.py) and current water quality data (get\_qualityData.py) were extracted from JSON files identified on the [beach report card](https://beachreportcard.org/) site. Requests were used to obtain the contents of each JSON file and relevant was data extracted.



Data Extraction Over Time

In a production environment, the CA Beaches data would be scraped, perhaps, monthly. The beaches and associated attributes are not expected to change much at all, over time. Each time this processing is run, it first deletes all rows from the beaches table, and does a complete insert of new data.

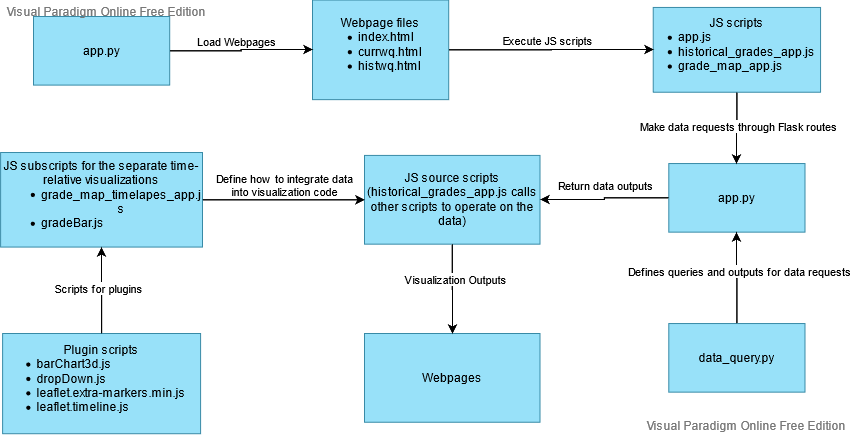
The historical water quality data will be extracted from the [historical water quality file](https://admin.beachreportcard.org/api/grades) once. The process deletes all rows in the grade\_data table and does a clean insert of all data extracted. It is intended to initialize our application data, and any subsequent data will be extracted from the current water quality file.

The current water quality data will be extracted from the [current water quality file](https://admin.beachreportcard.org/api/locations) daily. The json\_id and grade\_updated fields extracted from the file will be compared to the database, and if the combination of these fields does not exist, a row will be added to the database. In this way, we will be able to preserve grade updates and track changes over time.

### Overall App Process

#### Overview

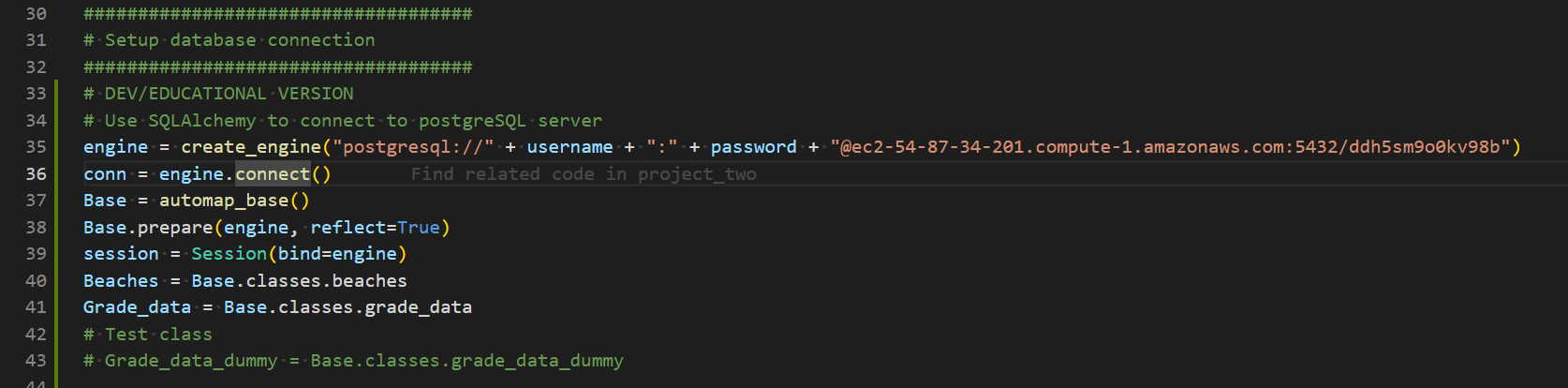
The entire app begins with the execution of a Flask app defined by app.py, with front-end routes delivering html files for web browsers. The html files execute JS scripts that in turn execute d3.json() requests through API Flask routes defined by app.py. These routes execute functions defined by data\_query.py which make requests to the postgreSQL server, returning back objects to app.py. App.py then converts the objects to JSON objects and sends them back to the JS scripts. These scripts perform further cleaning and process the code into visualizations, which are then sent back to the html with Leaflet. A diagram below illustrates the process flow:

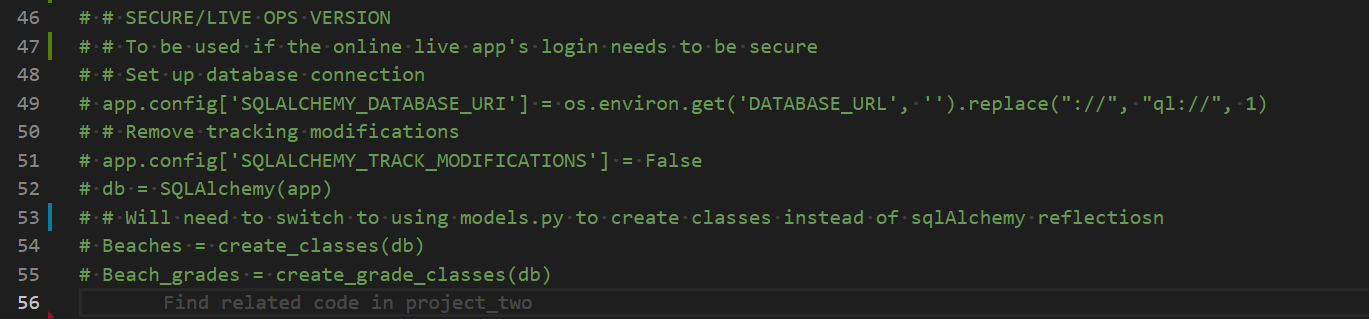


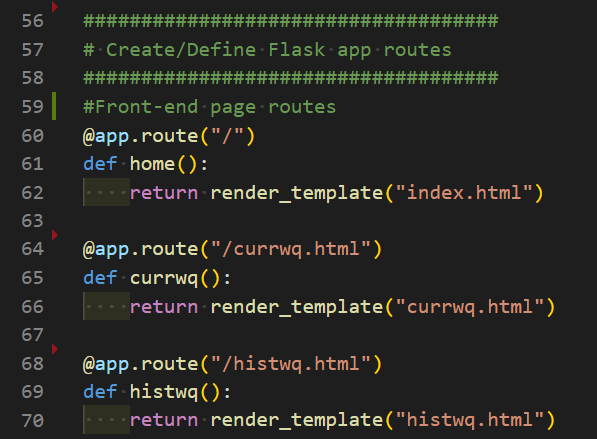
#### app.py

app.py serves as the hub to connect browsers to the HTML scripts and the JS scripts to the queries to be executed.

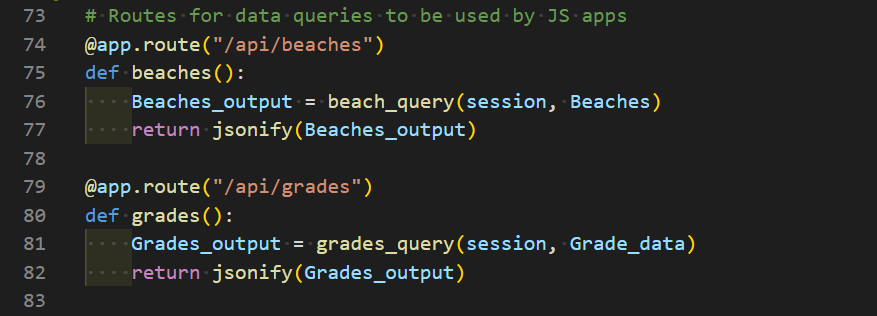
Connections to the SQL server are made using sqlAlchemy:



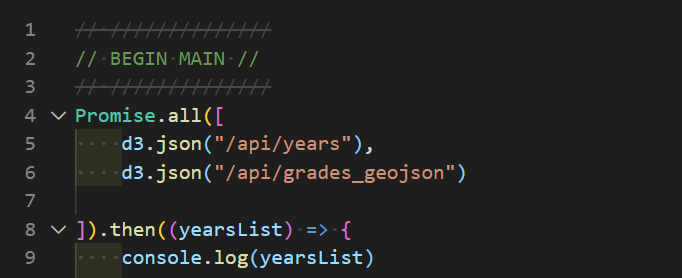
\*In order for guests to view the app while hiding login credentials within the code, the connection will have to use app.config and os methods to retrieve login credentials within the running environment to securely connect to the postgreSQL server. The process to reflect the tables as objects will have to be changed as well, utilizing models.py.  


Front-end page routes are defined, returning html for each route:  


Back-end API calls are defined, passing in the connected session and class objects that reflect the SQL tables. These calls will return jsonified versions of the query outputs.



#### JS Source Scripts (API calls)

JS scripts execute a d3.json() call, passing in routes defined from the Flask app:  


Scripts that make calls for data are the following:

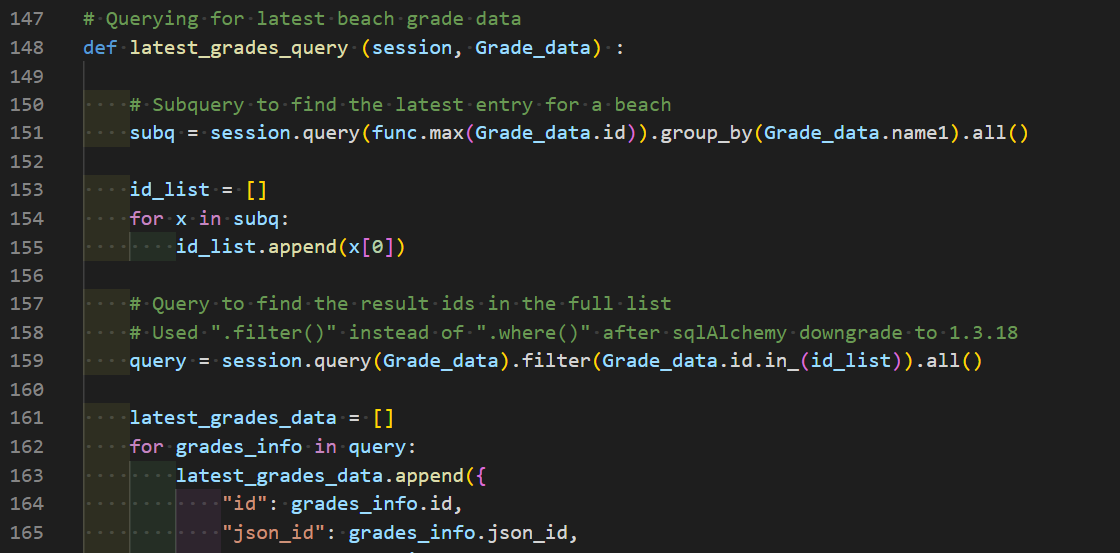
* app.js
* historical\_grades\_app.js
* grade\_map\_app.js

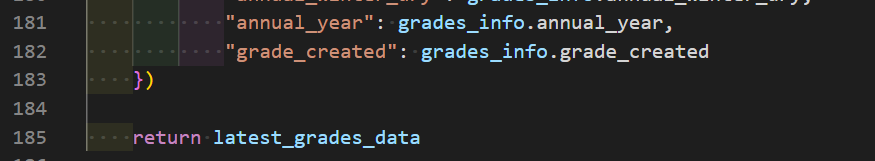
\*app.js and grade\_map\_app.js also operate on the data within the same script

\*\*historical\_grades\_app.js will call functions from other scripts to operate on the data

#### data\_query.py

data\_query.py holds functions that define what queries to make through the sqlAlchemy session and how to format the data to be jsonified through app.py. A function that executes a query and an output is defined for each visualization to be made.

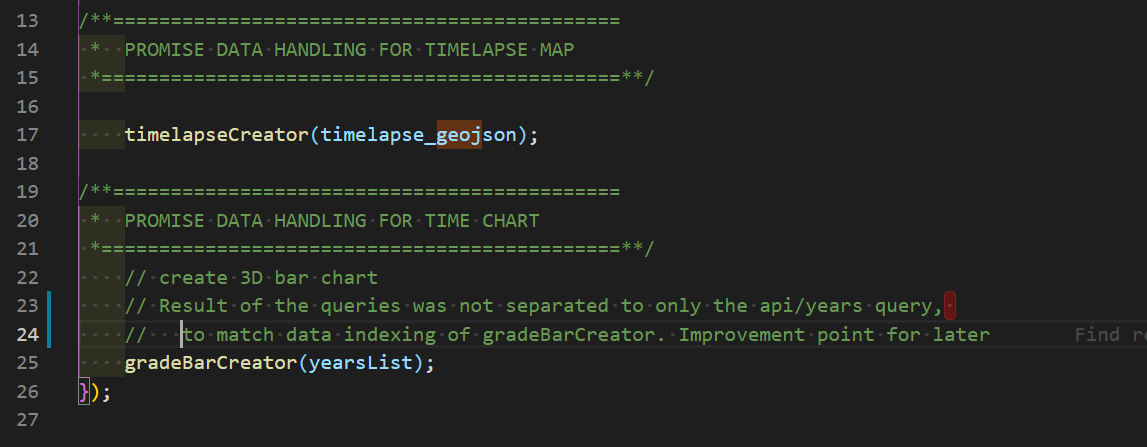


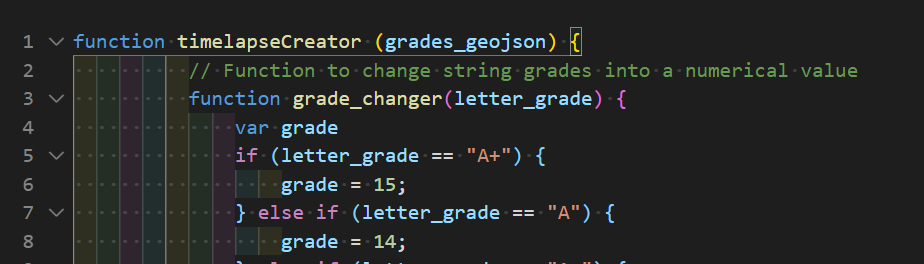
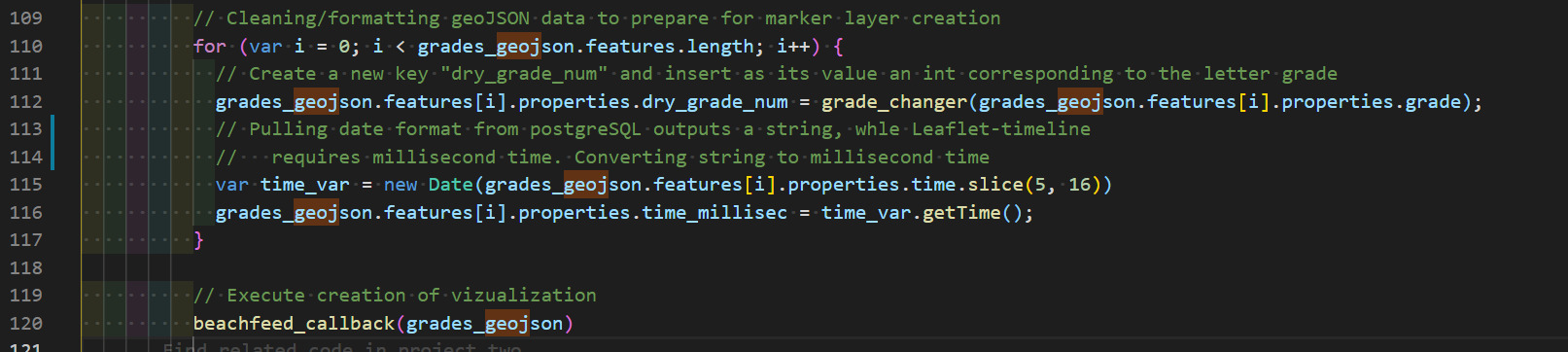


#### JS Source Scripts (generating visualizations)

The data will be returned back to app.py to be converted to a JSON object, and then retrieved by the JS code requesting for the data in its d3.json() call. The data will be operated on by JS scripts and visualization plugins, explained further through this report.

(e.g. historical\_grades\_app function calls that will operate on the data)

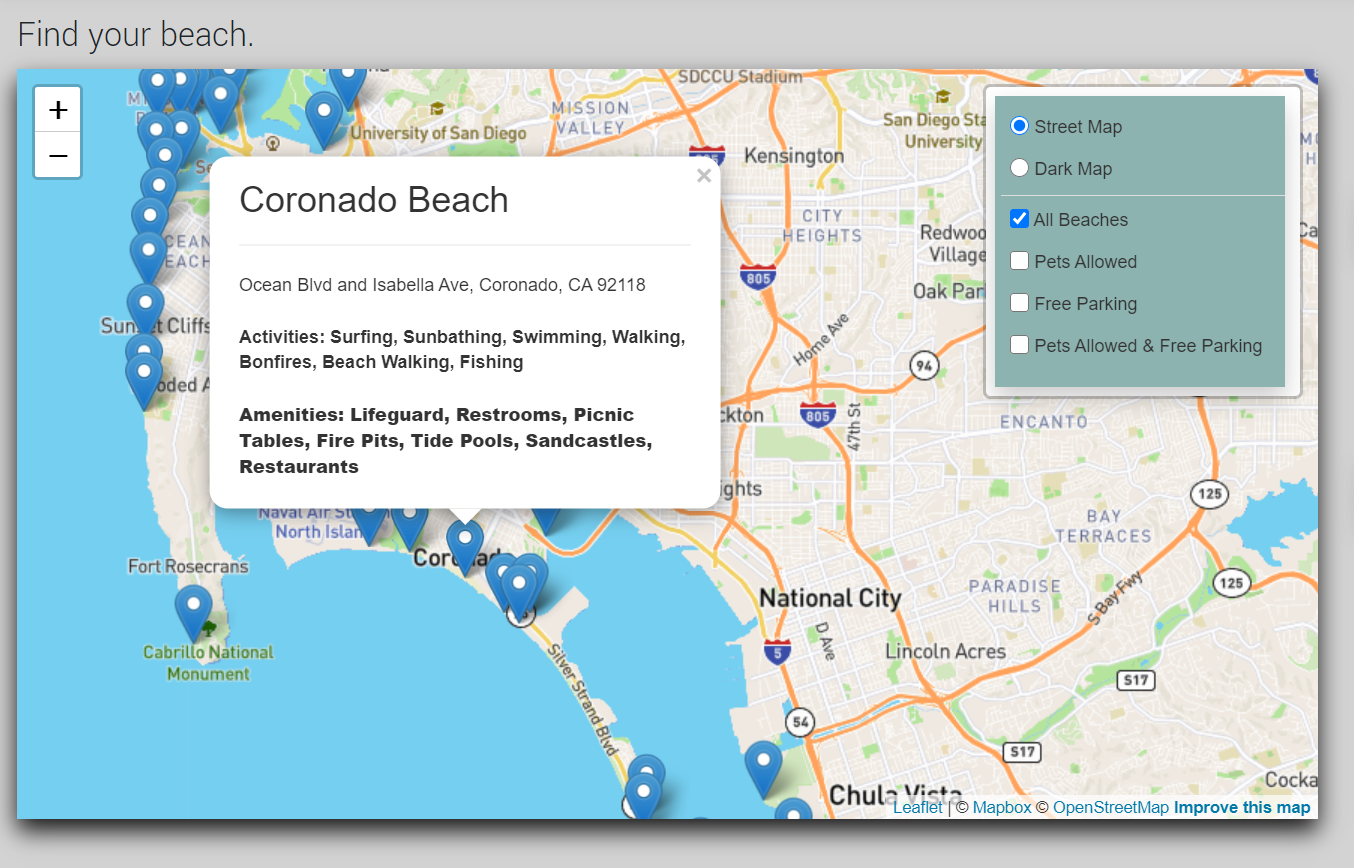


(e.g. examples of function definitions that operate on the data from grade\_map\_timelapse.js)  
  


### 

### Leaflet: Beach Information Map

The first visualization built was the beach information map which loads on the landing page. The process required pulling in the data via the flask route; the flask route is what connects the database to a python file by doing a session query to the beaches database. Once in the app.js file for the beaches information map, Promise.all was used to reference the flask app route for the beaches data, and a d3.json was connected to that argument in order to actually read in the route before creating a function to ultimately build the map. Once the database was connected successfully, 4 empty arrays were created to hold data for layer groups; one for the beaches, one for beaches with pets allowed, another for free parking, and the last array being for beaches that have both free parking and allow pets. Next, the data was looped through and iterated in order to reference each array and use L.marker to grab the latitudes and longitudes before binding a pop-up for each marker. The pop-up for any beach clicked provides the beach name, address, activities, and amenities. These pop-ups were pushed to each array and then set as layer groups so they could be referenced once the map was being built. By utilizing L.tileLayers for the basic map layers, the base maps were created and overlay maps were set equal to the 4 arrays created so users could have the option to toggle between viewing all beaches, just beaches with free parking, just beaches that are pet friendly, or beaches that have both free parking and allow pets. On load, the map has all beaches selected, as well as the basic street map tile layer.

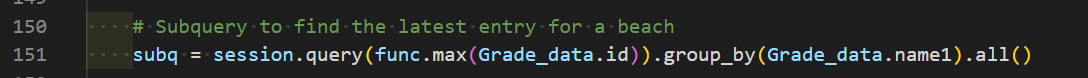


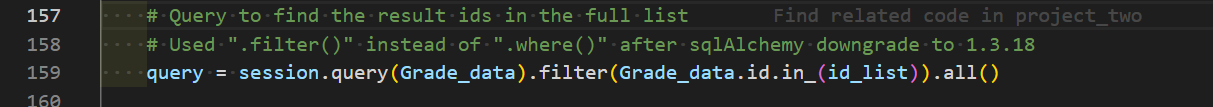
### 

### Current Grade Data Map (using Extra-markers plugin for custom markers)

This visualization is opened on the Flask route "/currwq.html" and its generation is defined mainly by grade\_map\_app.js.

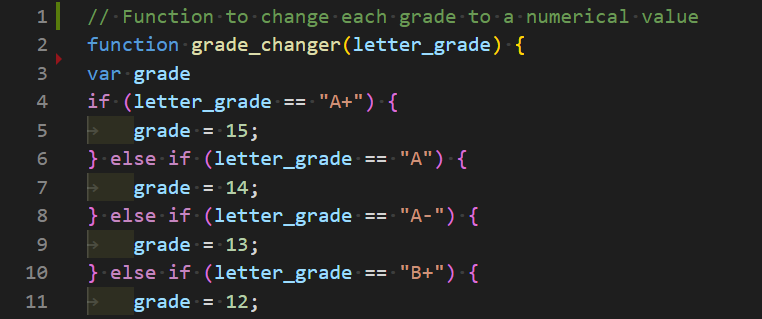
#### data\_query.py

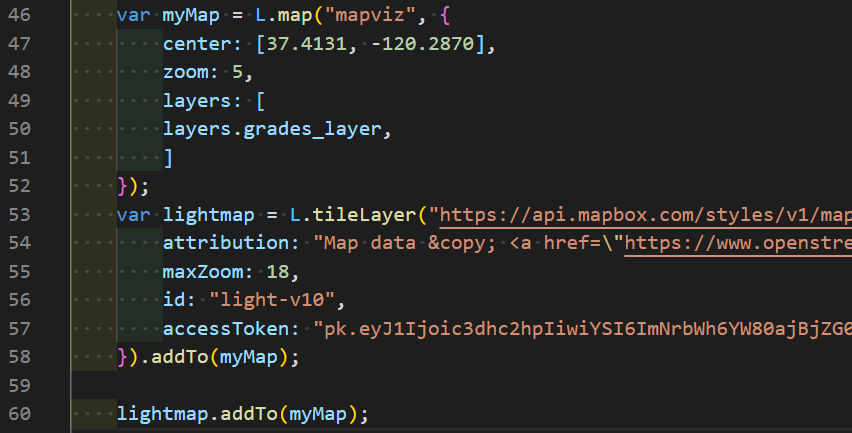
A query is made in data\_query.py to retrieve the latest grade recorded for each beach. As there are multiple entries for each beach, a subquery is made looking for the largest ID number (assuming the larger ID numbers are the "later records") for each beach name in the grade\_data table.  


A query was made for each ID number in the subquery's result to get the latest record for each beach.  


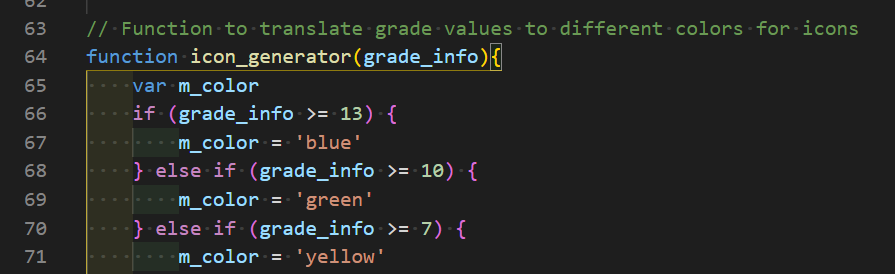
#### grade\_map\_app.js

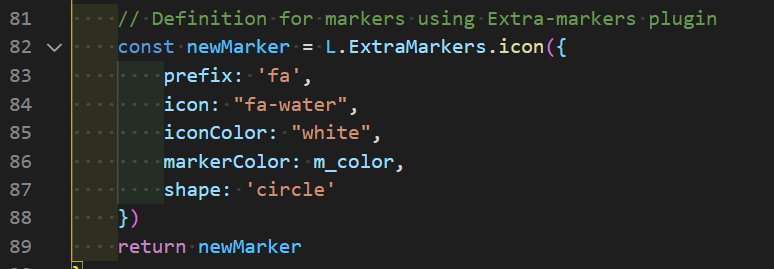
grade\_map\_app.js retrieves the required data with a d3.json() call, defines the functions it will operate, and executes said functions to generate the visualization.

A function to translate grades into numerical values is first defined:  


Map elements that are not dependent on the data are then defined:  


A function defining the attributes for each marker is defined:

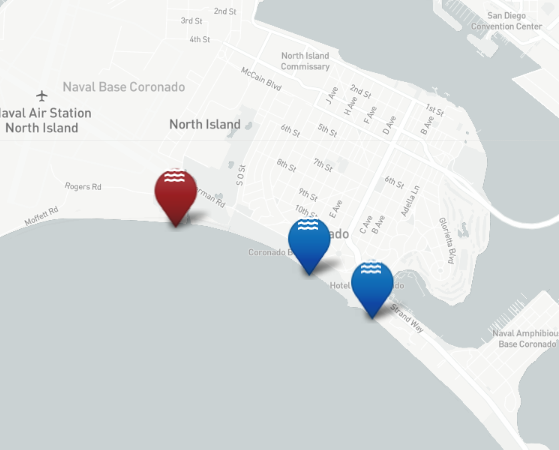


The function defines the attribute structure for each marker with L.ExtraMarkers, calling in a custom icon and changing colors.  


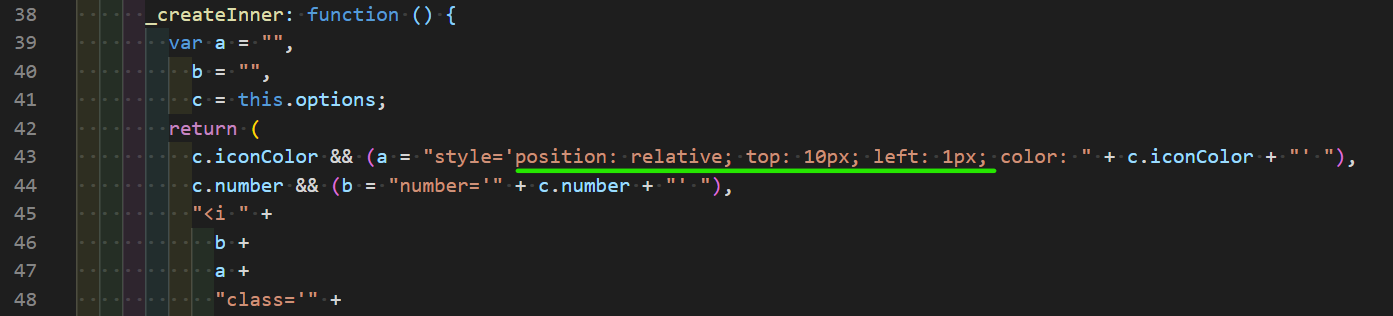
The script then executes the d3.json() call, operating on the data within a promise. The data is looped over, first converting letter grades to numerical values, and then sending the beach's numerical grade value into a variable to be referenced during the marker creation.

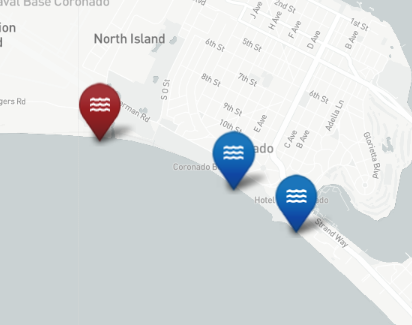
Line 107 begins the marker creation, using icon\_generator from the Extra-markers plugin, referencing the grade attribute to determine the marker's color.  


One adjustment was made to the Extra-markers plugin script, as the wave icon on the marker was too high and needed a positional adjustment.

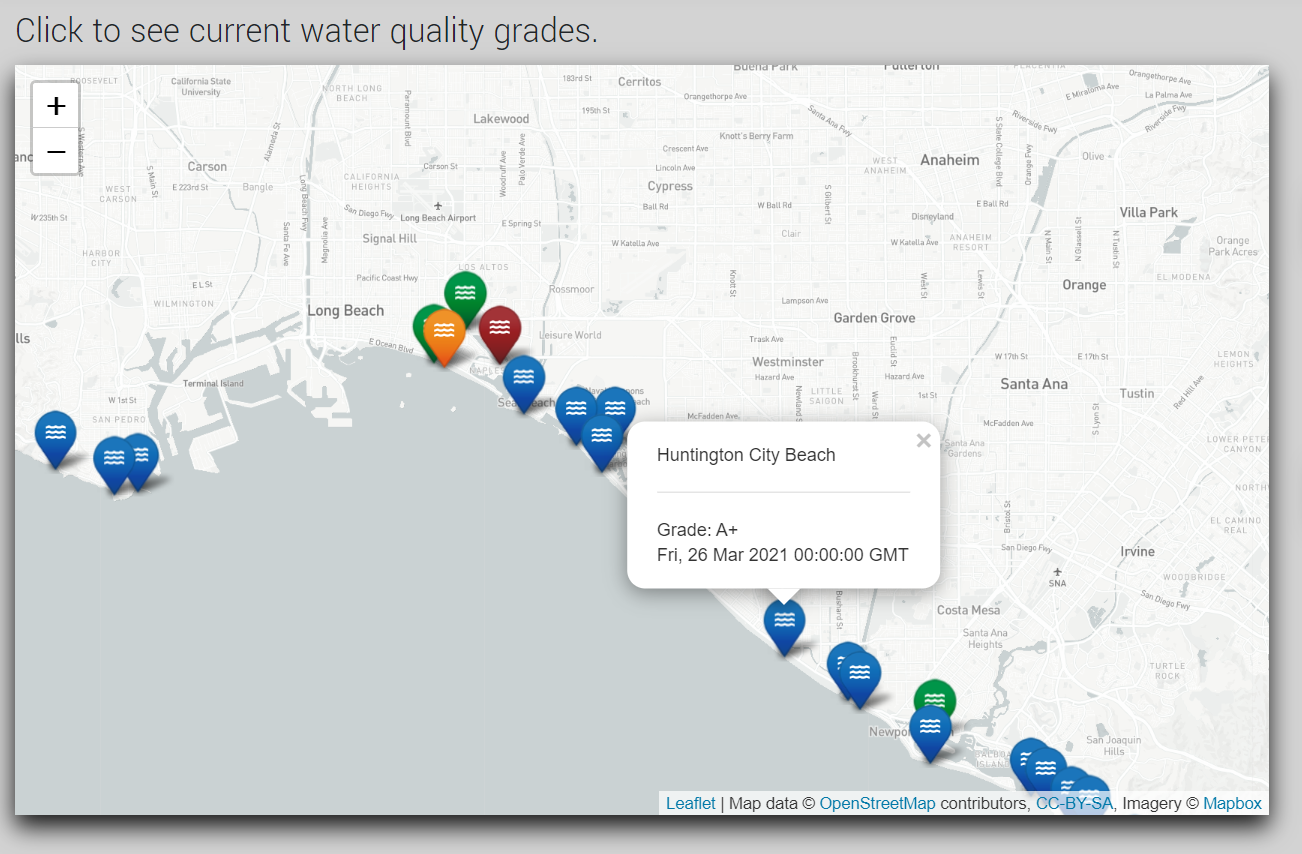


CSS styling was hardcoded into the icon generator, pushing the icon position lower relative to the marker's anchor.



The fixed position:  


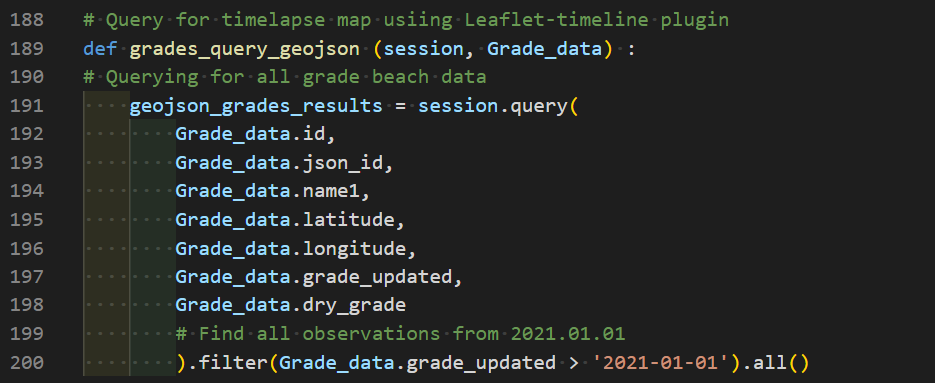
The visualization is then sent to the html div element "mapviz" for display.

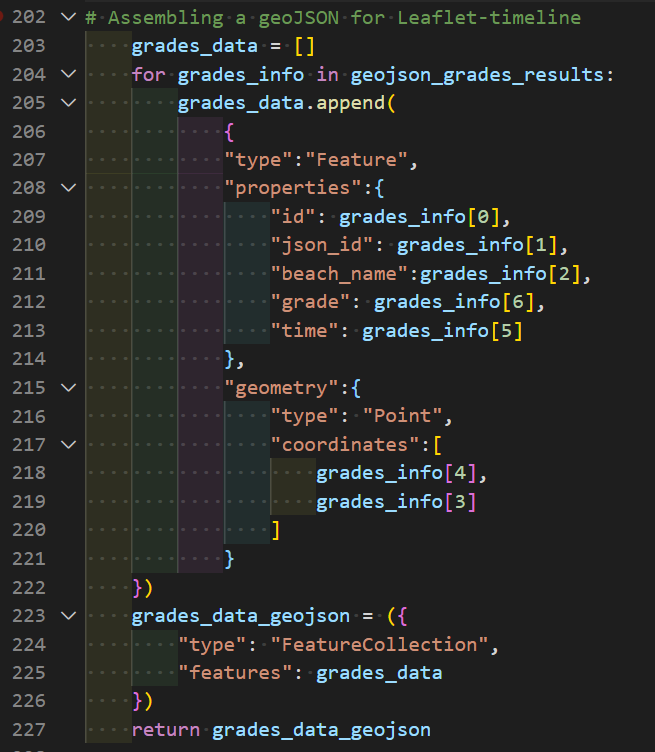


### Leaflet-timeline: Historical Grade Data Map

This visualization is opened on the Flask route "/histwq.html" and its generation is defined mainly by grade\_map\_timelaps.js.

#### data\_query.py

A query was made for select attributes, filtering for all data past a specific date.  


The Leaflet-timeline plugin requires a geoJSON object as an input. A geoJSON format was constructed within the grades\_query\_geojson function.  


The data is then passed back through the Flask app, which returns it to historical\_grades\_app.js to be operated on.

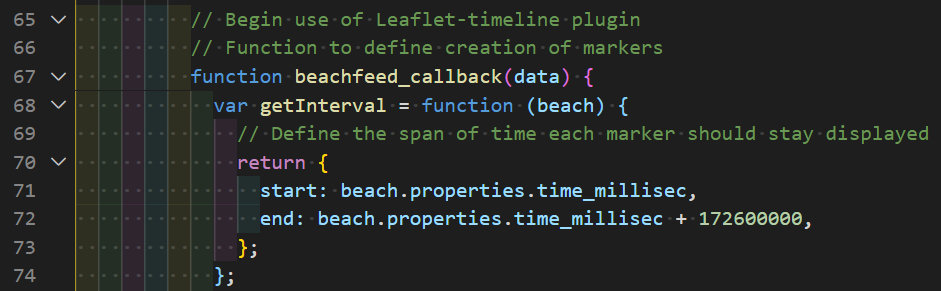
#### historical\_grades\_app.js

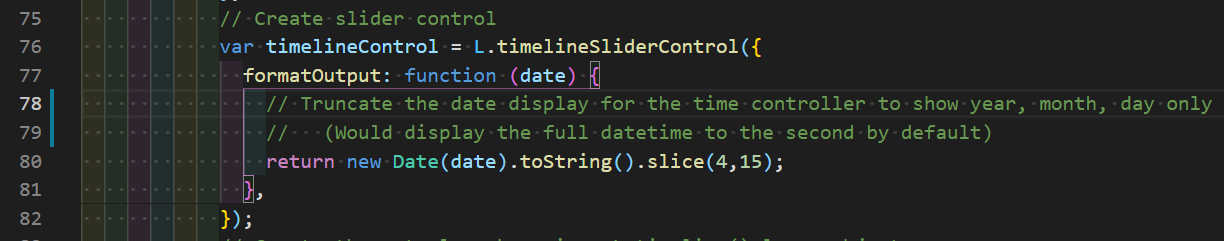
historical\_grades\_app.js makes separate calls into the same promise, one for each visualization to be displayed on the page. The data is called into the array, which is then separated through indexing for the function timelapseCreator() to operate on.

#### grade\_map\_timelapes\_app.js

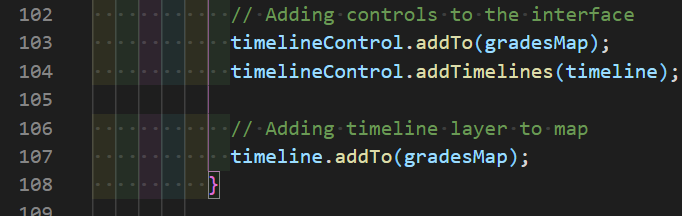
grade\_map\_timelapes\_app.js defines timelapseCreator(), which is the function that will generate the time lapse visualization.

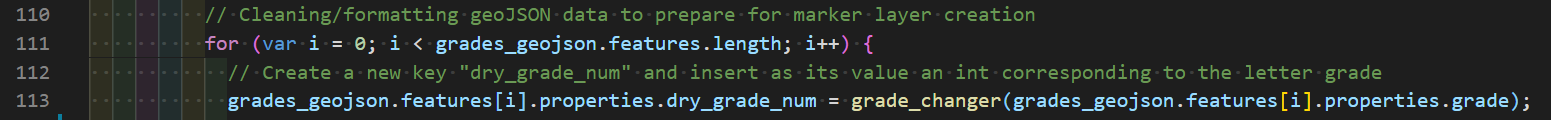
After changing the letter grades into integer values and preparing map elements that do not interact with the data, the Leaflet-timeline plugin is utilized within the function beachfeed\_callback(), which will ultimately generate the final visualization.

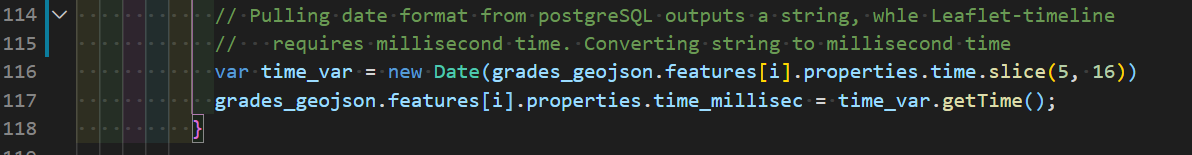
The time interval for each marker is first defined. This value needs to be in milliseconds time, which will be generated later in the code.  


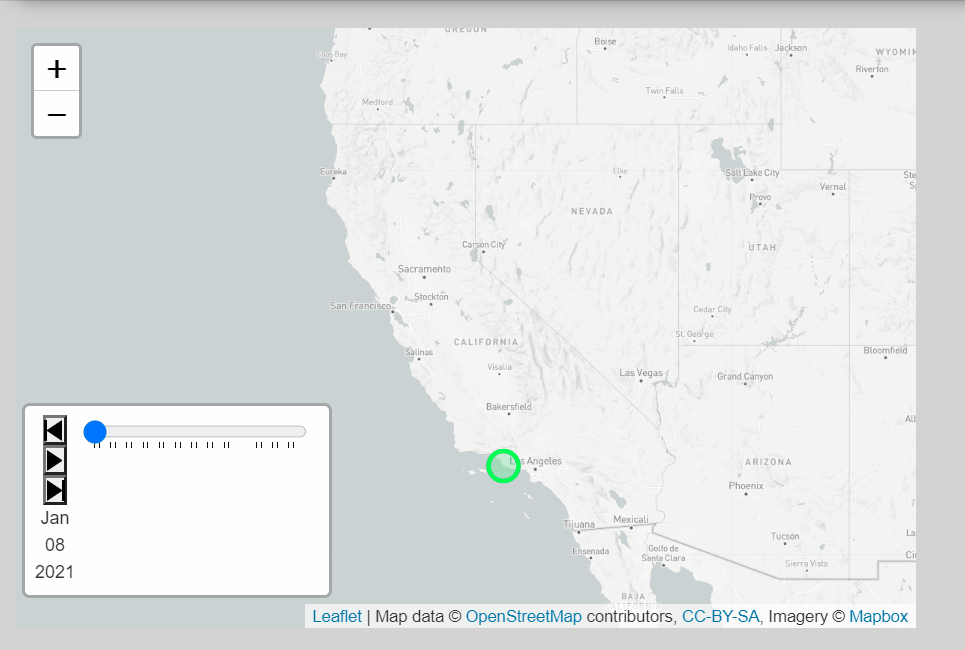
The user-interface slider is then created. The time will be displayed on the slider control itself which is the value being returned to the formatOutput key.  


Then the creation of each marker is defined, using the plugin method L.timeline() and defining its attributes:  


Timeline controls and the layer itself are added to the map.  


The code then turns to the execution of processing the data, first cleaning and formatting to prepare for the layer creation. First, numerical grades are generated based off of the letter grades:  


Then the datetime information for when each grade was recorded is translated to millisecond time:  


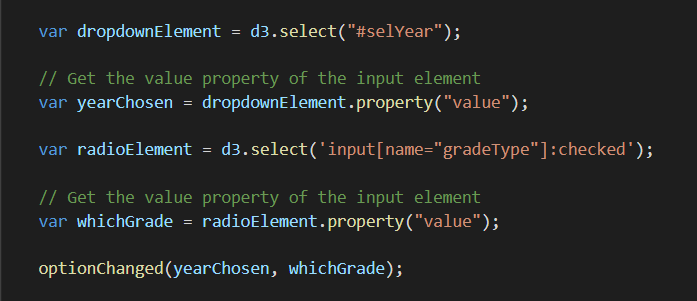
The visualization is then sent to the html div element "grades\_time\_viz" for display:  


Along with playing through the data chronologically, the slider can be used by users, click-dragging it to look at a specific day.

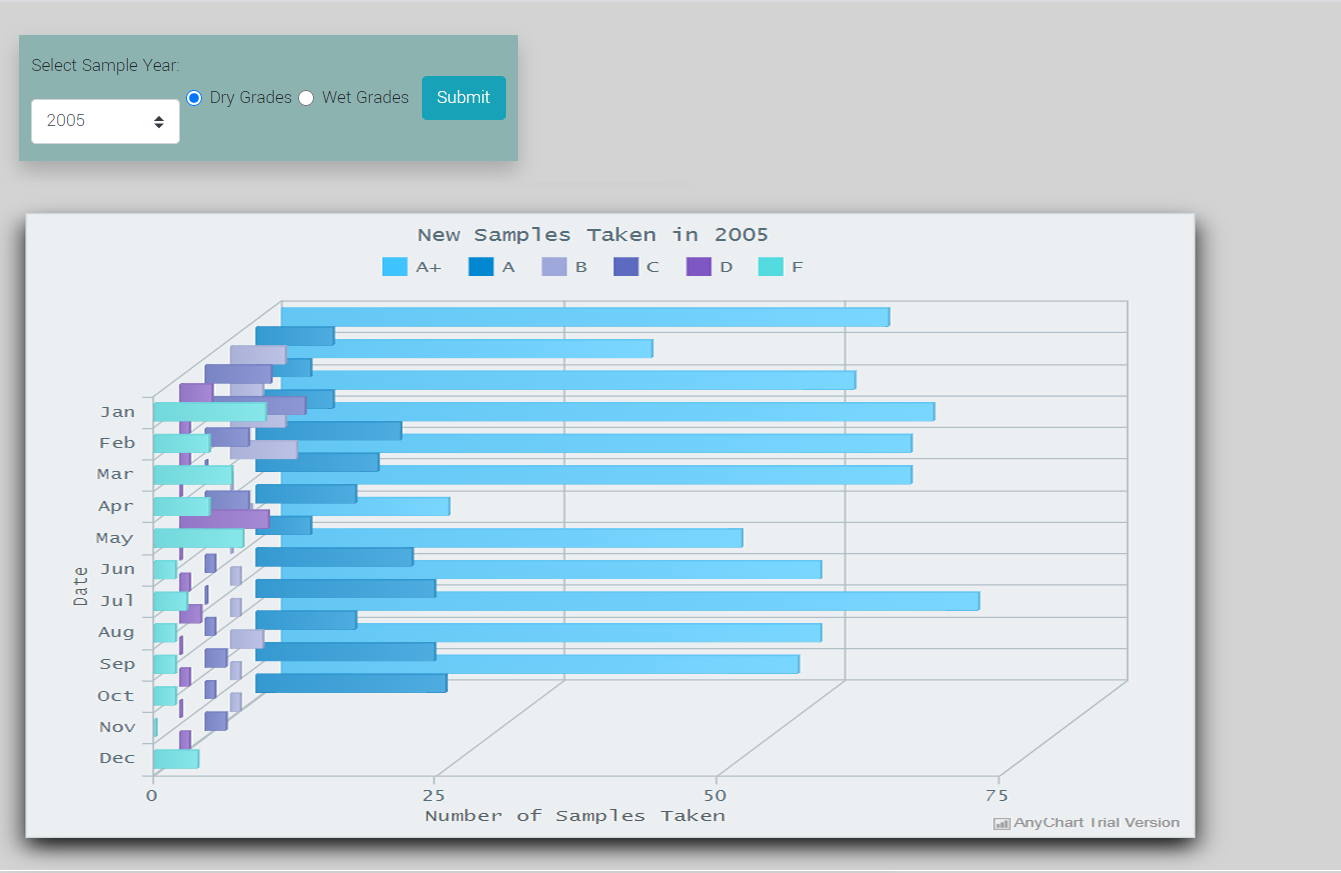
### AnyChart: Current Grade Data Stacked-Bar Chart

A 3D Bar Chart was generated using grade data to show a count of new water quality samples taken over time.

An SQL query was designed to first pull unique year data from the database in order to create a dropdown menu for the user to filter data for the bar chart. Radio buttons are also provided to allow the user to choose to display counts of dry grades or wet grades.



Based on the user’s selections, the database is queried for the requested data, and the bar chart is formatted and displayed on the page using id “viz3”.



### Building a Dashboard - Refining the HTML

A simple UI was built to display our visualizations. The landing page contains the CA Beach data on a Leaflet MarkerCluster map. The navbar and right sidebar have links to the water quality visualizations.

The current water quality map is displayed on its own page.

The historical water quality page contains a map showing changes in water quality over time, and a 3D bar chart showing counts of new samples taken over time.

Each page links to the others, and clicking on the brand marker will take the user back to the landing page from anywhere.

### Heroku - App Deployment

The GitHub repository was connected to the project's Heroku app space through the Heroku web interface. The repository's "main" branch is set as the deployment branch and is currently deployed manually under the following URL:  
<https://project-two-jh-ks-sw.herokuapp.com/>