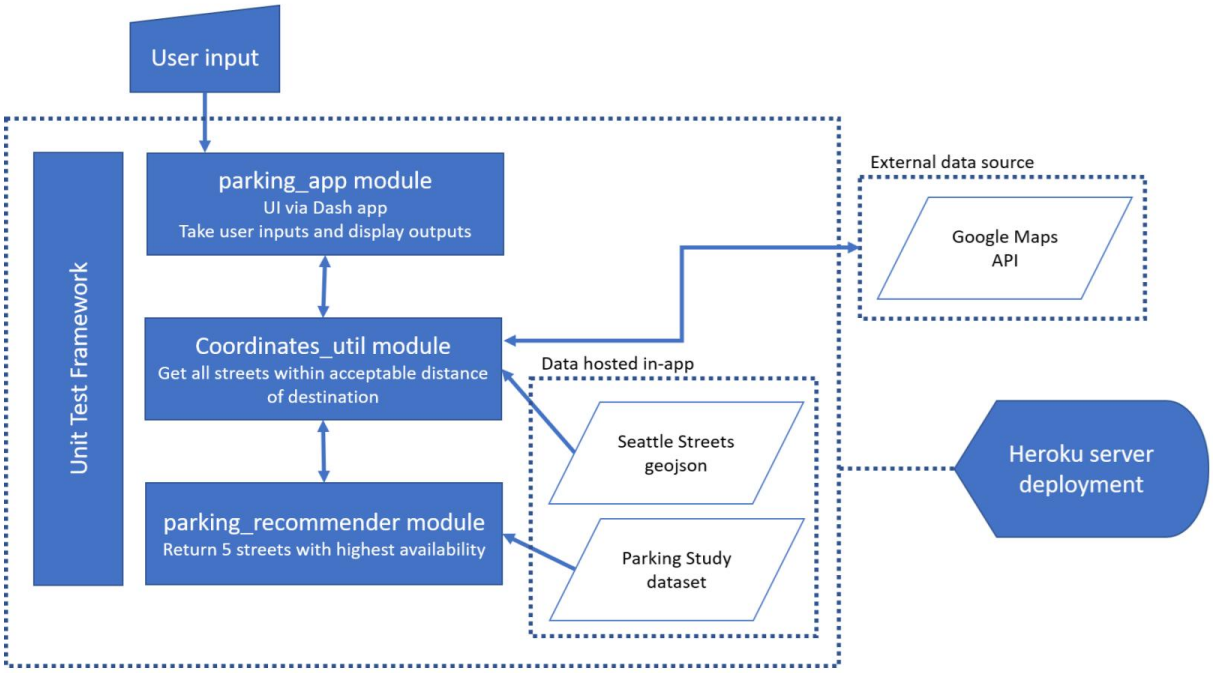
Component Specification

Data

How did we incorporate the data sources into our application?

Our application uses two sources of data, the Annual_Parking_Study_Data.csv and Seattle_Streets_MapDisplay.geoJSON. The geoJSON file contains coordinates to draw a line for every street in Seattle. We used the geoJSON to extract the corresponding lat/long for the streets that are in the parking study. Our application then uses the parking study data to calculate the average occupancy rate for the streets within the specified distance to the user's destination and the top 5 suggestions for where they should look for a parking spot.

Functional Block Diagram



UI

What is the user seeing and how do they interact with it?

On the dashboard, the user will see:

1. Input box, which will take the destination address and acceptable distance from the user;
2. Submit button to submit the details entered by the user;
3. Map box to show the recommended parking streets.

Once the user opens the app homepage, they see the input box and submit button on the left side, a clear Seattle map on the right side and the Seattle Parking title at the top of the page. The user can type the destination address in the destination inbox, the acceptable walking distance in the acceptable distance inbox, and then click the submit button. The map will be refreshed with 5 recommended parking streets in highlighted green. The user can hover over each recommended street to find the street address with an embedded Google Map link, distance from the destination and the number of spaces available. If the user clicks the Google Map link, it will launch Google Map experience to provide the user with directions to the intended parking spot.

Prerequisites:

1) Install plotly, dash, and pandas on the system.

2) To be able to draw a map via plotly, we also need a token from Mapbox, which provides various nice map styles.

UI Details

The general principle in building this dashboard via plotly and dash is to:

1. arrange and combine different elements together on a defined canvas
2. compile all elements in one container:fig=go.Figure(data=maps, layout=layout)
3. pass this container to dcc.Graph, in which dcc is the dash core components, and dash will create a html-based web application for the dashboard.

plotly.graph_objects package is used to create the scatter map by setting the map center as Seattle. The map has its own features within Layout, which was assigned to the variable mapbox, the numbers following mapbox and x/y is

arbitrary. Then within the Layout settings, we do adjustments for these two coordinate systems individually (such as the position of layout in the dashboard via domain, the appearance of ticks on each axis via showticklabels, and the ascending order of bar via autorange).

dash_html_components and dash_core_components are used to initiate the dash app. There are two containers. One container is for the input box and the submit button, which is passed to dcc.Input to create the input box and button. The other container is for the map passed to dcc.Graph. Inside of dcc.Input and dcc.Graph, we adjust the appearance for each container individually. For example, we restrict the input type of acceptable distance to decimal or integer only via "pattern". The number of clicks will be recorded to indicate when the map should be refreshed via "n_clicks". Then each of the containers is set to be shown proportionally to the size of the browser via "style."

In order to update the children of the "Output" component on the page("seattele_street_map") by defining Input, Output and State from dash.dependencies package, the @app.callback defines a function in Dash to be invoked when the value of the "Input" component changes. Whenever the n_clicks changes, the function that the callback decorator wraps will get called automatically. Dash provides the function with the new value of the input property as an input argument, and Dash updates the property of the output component with whatever was returned by the function.

The number of clicks (n_clicks), destination and acceptable distance data will be passed to the function submit_data(). Inside of the function, it will call get_parking_spots() from CoordinatesUtil class, which will return a list of objects that contains information about the 5 recommended streets. The list is then passed to Output("seattle_street_map"), which updates the map with the recommended streets. Then for each recommended street, hover information via figure shows the street address with an embedded Google Map link, distance from the destination and the number of spaces available. Once user clicks the Google Map link, it will launch Google Map experience to provide the user with directions to the intended parking spot.

Server

What is happening ‘under the hood’?

We used two main classes to build the application. Together they call the Recommendation Engine and provide data to the UI to display the information to the user.

Class CoordinatesUtil has the following methods:

- cal_distance calculates the distance between two coordinates in lat/long format
    - *Dependency:* Uses the Haversine library to calculate the distance
- get_destination_coordinates returns the location in lat/long of an address entered by the user
    - *Dependency:* Calls the Google Maps API to get the coordinates
- get_parking_spots returns a list of the 20 closest streets to the user's destination
- sea_parking_geocode opens a geoJSON file containing only the Seattle streets that are included in the parking study and returns a dictionary in which addresses are the keys and lat/long coordinates are the values

Class ParkingSpot has the following constructors:

- distance: the calculated distance between destination and this potential parking spot.
- coordinates: the coordinates of the start and end of this potential parking street.
    - e.x.: If the start and end coordinates of the street are (1,2) and (3,4), where 1 and 3 are latitude and 2 and 4 are longitude. Then the coordinates field will look like [[1,3], [2,4]]. This is the format which plotly uses to show a street on a map.
- street_name : the street address.
- street_lat_mid: the latitude of the middle point of this street.
- street_lon_mid: the longitude of the middle point of this street.

Recommendation Engine

How do we choose which streets to suggest?

The recommendation engine function is contained in the class ParkingRecommender. The ParkingRecommender object is initialized with the following arguments in its constructor:

- List of ParkingSpot objects returned by CoordinatesUtil.get_parking_spots
- Datetime string corresponding to the time the user submitted the request, assumed to be the time of interest in the query

It then passes the ParkingSpot list to its self.initial_list attribute, and parses the datetime string to extract the hour value and passes it to its self.hr attribute. The __init__ function also calls the self.slice_df() method, described below, to filter the full Seattle Parking Study dataset and pass the filtered dataset (as a Pandas dataframe) to the self.initial_df attribute.

The ParkingRecommender class contains four methods in addition to __init__:

- slice_by_street() imports the Seattle Parking Study dataset from a csv file and performs slicing operations to reduce it to include only observations of the streets in the ParkingSpot list.
- slide_by_hour() further filters the dataframe for the hour of day specified at the time of the query. This method is called in __init__ to pass the filtered dataframe to the self.initial_df attribute.
- max_freespace() takes the dataframe from self.initial_df and calculates the estimated number of available spaces for each of the streets of interest.
- recommend() calls max_freespace() and fills in the ParkingSpot.spaceavail attributes (which were initialized to 0 upon object construction) for each object in the list with these estimates, and returns a list of the 5 ParkingSpot objects with the highest estimated availability.