DSCI 551 Final Report

**Description of the User-Interface**

In implementing the front-end portion of our application, we decided that each of our three pages, ‘Mission Statement’, ‘Specific Parking & Availability Updates’, and ‘Locate Nearby Parking’ would each have their own function that when called, would render the page and its features. Furthermore, for specific functionality related to the backend, such as updating availability or querying for locations based on index, functions representing them in the backend are imported over and nested within each page’s function as needed. Users are first presented the Mission Statement page by default, and on the left, there is a radio widget which allows users to navigate to other pages of the app by clicking on the icon associated with it. On the homepage (Appendix Image 1), there is a section dedicated to explaining our motivation for building Park It, along with a background on the data used and a dedicated section explaining the 3 core functionalities we provide.

The remaining two pages the user can navigate to contain the core functionality of our application. On the Specific Parking and Availability Updates page, users are first presented with a form and prompted to enter a parking spot’s specific address. In the scenario that a user already knows the address of a parking location and its type (Street Parking, Garage, Lot, Not Street Parking), they are able to fill out the form with these details and click the search button (Appendix Image 2). These user inputs are then passed as parameters to a backend function, location\_description( ), which queries our database on those values to return various strings corresponding to different types of information on that location. The user is then notified that their query is loading while they wait by using a st.spinner( ) function that disappears once a function is finished running.

Once the query is complete, its contents are printed out to the application containing the address, type of parking, valet status, safety score, availability, and price (Appendix Image 3). Below this, two options for the user are available once they finish looking at the specific parking information: a checkbox (st.checkbox( )) to clear the query results, and another checkbox to proceed towards the availability update function of our application. When the latter is clicked on by the user, they are then prompted to enter the availability for that specific location they looked up. The assumption is that a user may have looked at the information for a parking spot marked as available, but when they drive there to look for parking, all spots are occupied. Therefore, a user can notify other drivers that there is currently no space available in real-time by selecting occupied from the drop-down menu. Once selected, their response is sent to the backend to update the availability value for that location in our Firebase database to match what they input. They are then notified of this change if successful, and they are prompted to then click on a checkbox to clear the query result. If a user re-submits the query the changes made to the location’s availability status are made (Appendix Image 4). Whether a user decides to update availability or not, they are prompted in both scenarios to clear the query result as not only is it more appealing visually afterwards, but it is also important to certain functions of the Streamlit library that are later discussed in our section of the report explaining the challenges associated with using Streamlit.

The final page of our application is the Locate Nearby Parking page, which allows a user to input their location and desired filters to find parking locations that fit their criteria (Appendix Image 5). Like the previous page, the user is presented with an identical form to fill out an address, except instead of a location’s address they input theirs. The user is then prompted below to select any filters they might have as requirements for locations that could be returned as results. These filters range from parking type, valet preference, restrictions, availability, and a desired safety score. In addition to this, users can select how to sort the locations that match their criteria. Commonly available sorting criteria, such as distance and duration, are options for the user, but keeping in mind our motivation to help driver safety, we also allow user to sort locations by its safety score. Finally, the user can select how many results they would like to view from 5-25 in increments of 5.

Once the form is filled out, the user clicks the search button and like the previous page, gets a notification that their query is loading. Their various responses are then stored in variables and passed to the appropriate backend function. In this case, the parking\_list( ) function is called first, which queries the database and sorts the results based on user criteria, then returns a dataframe of the results and a dictionary containing the coordinates for the user’s location. These values are then passed into the Map( ) function for our frontend which uses pydeck, a high-scale spatial rendering library, to display a map to the user. This map is done in a scatterplot style with two layers. The first, is where the user’s longitude and latitude values from the dictionary are plotted as a green point on the map. For the locations in the dataframe, the columns containing latitude and longitude data for each instance are passed as data for the map, and they are plotted as red points on the map as the second layer. The map is interactive and allows the user to zoom in/out and drag the map around (Appendix Image 6). Unfortunately, we were not able to include labels for points on our interactive map, as it required an html template and other configurations to implement, which would potentially increase the load times for our application.

At the same time the map appears, a table representing the dataframe of locations is also outputted to the user (Appendix Image 6). Each location has individual columns for address, distance from the user, the time it takes to driver there (duration), its safety score, and an ID value representing its index in our Firebase database. The latitude and longitude columns used for the map are dropped in the front-end code before displaying the dataframe, given that they are irrelevant for the user. Like the map, this table is also interactive to users and allows them to sort rows, one attribute on a time, by clicking on column headers. Once the user has looked at the results and decided on a location they would like to park at, they can find more detailed information on its pricing and hours in the drop-down menu below the table. Here they are prompted to select one location by selecting a number that corresponds to the index provided in the table. Once an index is selected, it is passed to a different version of the location\_description( ) provided by the backend, which takes in an index and queries Firebase for the information related to the corresponding location. This returns the same values as displayed in the Specific Parking Information and Availability page (Appendix Image 7). If a user decides to instead view another location, they can click on the x next to the index in the drop-down menu to clear their search to select a new index. Once a user views one location’s specific information, a checkbox appears below the information to clear the entire query result once they are finished looking at all locations they are interested in.

**Front-end Challenges**

In developing our application, we faced challenges with the Streamlit library that made it more difficult to develop our UI in the way we desired. One of the first challenges involved printing strings and other text to the application. Streamlit does not render formatted Python strings correctly, as newline characters (“\n”) in a string are required to be a 2-space distance away from the actual content that you wish to put on a new line to work properly. Tabbing/indenting lines with the corresponding python character (“\t”) are not rendered at all, and after extensive searching for potential solutions, only the solution for newline characters could be found. Unfortunately, this bug and its work-around are not mentioned within Streamlit’s documentation for the st.markdown and st.write functions, both of which print out text and data to the user. As a result, we could not format information outputted to the user in as formal of a design as we wanted

Another issue we encountered was with Streamlit’s use of the pydeck library for more interactive maps instead of the default st.map function. While pydeck allowed us to center the map by default to the user’s location, and control other aspects such as icon size and zoom, there were constraints when implementing multiple layers. To display text or other values when a user hovers over a point on the map, this required a specific html file to be referenced as the tooltip= value for the map. However, since one layer was for location data and the other layer was for user data, multiple html files would have to be specified as values, which pydeck did not allow. Given time-constraints for the app’s loading speed, and the complexity of integrating multiple html files with both the back/front end, this feature unfortunately could not be implemented.

The biggest challenge faced when using the Streamlit library was being able to maintain session state when working with button widgets to prevent pages from reloading and erasing user inputted data. As acknowledged by Streamlit itself, there are issues with its st.button function for button widgets. Prior to the button being clicked on by the user, all functionality and interactions made are saved onto the page. This is still saved even after the button is clicked; however, the caveat is that all functionality after a button press can only be non-interactive to preserve the page’s session with user data. For example, in the page for Specific Parking Information and Availability Updates, I originally had a user fill out a form, click on a search button, and then display the information for the queried location below. However, once we decided to implement an availability update feature, trying to add it as a checkbox below the displayed information proved difficult to use at first. This was because every time the user clicked on the checkbox to procced to the availability update function, it would reload the page and reset the query results that had been displayed up to that point.

After extensive searches on Streamlit forums to try and find the solution, it turned out that in one of their forum posts, they acknowledge that the button widget does not work as a developer using Streamlit would intend for it to. This is because the button is tied to a dictionary hidden from the user, which has a Boolean value corresponding to the status of the button. This is Streamlit’s way of managing session state, and by default, Streamlit always re-runs the entire code for a page once an input is received. In the case of the button widget, re-running the page sets the status of the button in session state to be reset to False, as if the user had not pressed it, even though they did previously. Because writing is the only Streamlit functionality that does not require re-running the page, initially only print statements could be performed after a button press. Therefore, to maintain the session on the page, the dictionary value for “button” had to be set to True permanently once a user clicks on it, so that even when a page is re-run, the value in session state remains True.

While this allows for unlimited user interactions and functions to run after a button is pressed, the value for the button must also be set to False once the session is complete. Otherwise, the saved session will automatically run every time the user refreshes the page, or when another user interacts with the address and parking type inputs before clicking search. This means that results will appear even if the user does not click the search button or has finished entering inputs. Therefore, to avoid a poor UI design for both pages with user interaction, users are prompted to clear the query result. If they click on the checkbox, then the value of the button’s state in the session state of Streamlit is set to False, and then any other interaction with the page will clear the session, allowing the user and others to properly interact with the page. In the case of our Locate Nearby Parking page, this had to be considered when deciding how to allow a viewer to select a specific parking spot from the many results their search returned. Due to the complexity of maintaining session state and its sensitivity to user interaction, it was decided that it would be better to have parking information for a location a user selects to display on the same page rather than redirecting to the Specific Parking Information page.

Other minor changes experienced with Streamlit include limited interaction in displaying dataframes as the interactive tables cannot be sorted on multiple column headers, and some columns refuse to sort properly. However, outside of the previously mentioned challenges, Streamlit provided a streamlined way for us to develop the frontend of our application and integrate with the backend functions by just using the Python language. While the library has several quirks and features that have not been developed as much as we would have liked during the creation of our application’s frontend, it allowed us to overcome the challenge of losing our previous teammate with frontend experience.

**Appendix: Images of Park It UI**

Image 1: Homepage with mission statement, boundaries of Park It’s operation, and description of core functionalities

A screenshot of a computer

Description automatically generated with medium confidence

Image 2: Specific Parking Information page structure, and sample user query

A screenshot of a computer

Description automatically generated with medium confidence

Image 3: Information retrieved from sample user query and subsequent availability update feature

A screenshot of a computer

Description automatically generated with medium confidence

Image 4: An updated availability status for the sample user location, and prompt for clearing user query results.

A screenshot of a computer

Description automatically generated with medium confidence

Image 5: Locate Nearby Parking page structure, and sample user query

A screenshot of a computer

Description automatically generated

Image 6: Interactive map and table displaying the results of the sample user query

A screenshot of a computer

Description automatically generated with medium confidence

Image 7: User search for a specific location’s information using its index value (ID column) from table + Option to clear query

A screenshot of a computer

Description automatically generated with medium confidence