2020

System Documentation

CWU PARKING APPLICATION

KEVIN BERTELSEN – CAMILO JACOMET – COREY JOHNSON – HAILEY LAWTON
PAUL MCCAFFERTY – DRAKE WALD

HAILEY & THE BOYS | 400 E University Way, Ellensburg, WA

Table of Contents

Introduction

A considerable issue on the Central Washington University (CWU) Ellensburg campus is a lack of accessible on-campus parking. This is a problem for students, staff, Ellensburg locals, and other visitors to the university. This problem only continues to grow as the student population continues to increase every year. In order to solve this problem, Dr. Szilard Vajda has requested the creation of the Central Washington University (CWU) Parking Application. This application will assist in guiding users to available on campus parking lots in a quick and efficient manner.

Initial Requirements

The initial requirements outlined for this application were as follows: a user-friendly application which uses real-time data in order to constantly inform users of the number of parking spots in each parking lot on the CWU Ellensburg campus. Based on the location of the user, the application should guide them to the nearest parking lot with available spaces. The application should run at all times.

These requirements have changed slightly throughout the process of creating the application, as additional functionality was added to the user interface and there were some complications when completing the object recognition for real time data collection.

Accomplishments

The current version of the application includes a user interface which is complete with filters for different types of parking lots, consistent and constant pulls from a database to update user information, directions from the user's current location based on user's parking lot selection, and redirection in the case that the parking lot fills prior to the user reaching their destination. The current version also includes a Python script, running from our Ubuntu server, which is able to push random values to the database at scheduled intervals in order to simulate real-time parking data.

Purpose and Overview

The purpose of this document is to describe in detail the software and hardware which have been installed and implemented in order to complete the current version of the CWU Parking Application. From this description, users should be able to recreate the application in such a way that the application is fully-functional. This will be accomplished in two parts: a description of the tools which have been implemented, and an accompanying documentation of the code that has been written.

Software System Attributes

Reliability

The application should display the most recent version of the data stored within the database to the user of the application correctly 99% of the time. The development team has strived to achieve this goal with thorough testing. This testing involved the manual placing of values directly into the database and watching the application update the values in real-time on the mobile device in use.

Availability

The application is available for use 24/7, as there may be individuals who require information regarding parking at any time of the day. We have increased the usefulness of this feature by allowing users to

filter the parking lots which they are displayed so that they are only shown information regarding 24 hours parking lots on campus.

Maintainability

The software is organized neatly into sections based on the function of the code at each point. These sections have been detailed below in the Code Structure section below. Data regarding each parking lot has been organized concisely within a local JSON file located within the data folder and entitled parking-lot-data.json, as well as the Firebase database which stores information regarding parking spots. This data is stored in such a manner as to make it easy to add additional parking lot data as more parking lots are added to the system.

There is one area of the code that will need to be altered in order to match a lotIndex with the lotIndexes which are listed within the database. A description of how this code should be altered will be included in the discussion of the code structure of the user interface below.

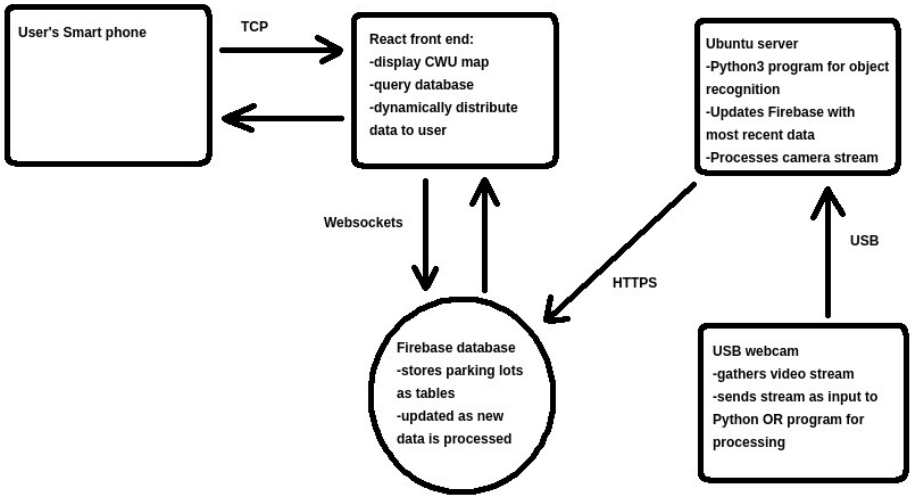
Feasibility

The current version of the system provides an overall proof of concept for the requirements outlined at the beginning of the project lifecycle. One of the requirements which will need additional work in the case that further development of the application occurs is the object recognition script. While a low level of accuracy has currently been accomplished through an API found on GitHub, the development team aims to achieve greater accuracy in future versions. This, accompanied with camera installations in parking lots across the CWU Ellensburg campus, will help ensure that the application is as useful to users as possible. We believe that with more work, and an investment in the proper hardware, this project has the capability to be a long-term solution that will benefit students, staff, and visitors alike.

System Design

High-Level Design

The high-level design outlined below describes an overview of the architecture that will be used for developing the application. This includes an overview of the entire system, as well as some of the tools which have been used in the development process. A more detailed discussion of the tools being used will be included in the Tools and Technology section of the document.



In the current version of the application, the USB webcam is not included. For future versions of the application, the development team hopes to include further implementation of the object recognition script which would utilize the webcam present in the high-level design. Details regarding the current version of the webcam and object recognition script will be detailed below.

Structural Design

The structural design outlined below describes an overview of the communication processes between backend and frontend of the application. The application requests that the user accept responsibility for their legal obligations to follow all rules of the road, and once they have entered the maps screen, the application begins communicating with the database in order to query data regarding the number of parking spots which are currently available in each parking lot. During this time, the backend will also communicate with the database in order to post values regarding the number of currently occupied parking spots in each parking lot.

Operating System Requirements

This application may be run on both Android and iOS devices. Currently, in order to run the application, the user must be able to run React Native on their computer in order to gain access to the application on their mobile device. React Native is available on Linux, macOS, and Windows operating systems. Throughout the development process, the front end of the CWU Parking Application has been built on Windows 10 Home operating system, and the backend of the system including the server has been built on Linux Ubuntu 18.04.

Tools and Technology

The following is a description of the development tools which the development team has been using throughout the software development lifecycle. The tools which have been included are React Native, Python, and Firebase. A brief description of the tool itself and how it is being used within the project, and an explanation of why the development team selected this tool above others.

React Native

React Native has been used in order to develop a cross-platform user interface which will run successfully on both Android and iOS devices. This tool is used by implementing JavaScript code while making use of the React Native JavaScript library. By running the code on an emulator or using the Expo mobile application for Android and iOS, React Native users are also able to track the changes that they are creating within the application in real-time as they save the work within their JavaScript files.

In order to install and use React Native, a user must first install Node.js. While the most recent version of Node.js was initially recommended to the development team, the use of this version caused some technical issues in the early stages of the application's development. In order to remedy these issues, the development team installed an earlier version of Node.js, specifically, version 10.16.3. This version has been shown to avoid these previously mentioned issues and is still available online for download on Linux, Windows, and macOS operating systems. Further details regarding the installation and use of React Native for the development of the application are included in the User Documentation.

The development team elected to use this tool in order to decrease the number of complications associated with creating a cross-platform application. While there were some specific components

within the JavaScript that needed to be changed in order for the application to run successfully within Android and iOS platforms, these changes were minimal. This tool was also selected in order to have relatively simple interactions with the Google Maps API which is being used in order to guide users to their destination, and Firebase which currently stores pertinent information regarding each mapped parking lot.

Python

The Python programming language has been used in order to develop the backend and object detection components of the application. Within the Python script, OpenCV is used in order to capture and gather images from a webcam, and an object detection API which has been implemented is used in order to count the number and types of objects which are found within the captured photograph. We have also implemented a test function which is used to randomly supply integer values to the database in order to test the user interface of the application.

The Python programming language was selected for the project due to the fact that it is a high-level object-oriented programming language which is currently gaining popularity among the software development community. Python is also able to easily interact with OpenCV and object detection APIs, as mentioned previously.

The server that we are running is a surplus server running Ubuntu 18.04, where the Python scripts are running natively every 30 seconds on a crontab job. Crontab is a Linux utility that allows for the scheduling of automated script executions set up at a given interval of time, and is highly-customizable. The version of Python that we are using is 2.7, which is what is compatible with OpenCV for the camera interfacing that is done in conjunction with the server.

Firebase

Firebase database has been used in order to store information regarding the number of total parking spots in each parking lot, as well as the number of parking spots which are currently occupied within the lot. Firebase operates as a real-time NoSQL database which is able to easily store and sync the cloud hosted application data. Firebase is especially useful due to the fact that instead of using HTTP requests, which can be laggy, it utilizes data synchronization, which means that every time data changes in the database, all connected devices update within milliseconds. The information within the database is stored in a JSON tree, meaning that Firebase users are able to quickly load files containing JSON trees to automatically create databases.

The development team elected to use Firebase due to the convenience of access and use. The value for the current number of parking spots filled is able to be constantly updated by the backend. In the current version, this is done using a Python script which is used to populate the database with random numbers. Due to the fact that the database is hosted on the cloud, more information regarding the parking lots will be able to be easily added without concern regarding the need for additional memory space. This is a considerable positive due to the fact that, if the application is expanded upon in the future, more parking lots will need to be added in order to track their information in addition to those parking lots already included in the database.

Code Structure

The following is a description of all components of the application code which are imperative to the recreation of the project. This description has been split into two sections, a section for the code involved in the development of the back end of the application, and a section for the code which is used to render the user interface of the application. The further separation of the code into manageable sections will be detailed below.

Back End Development

The following code structure description describes in detail the Python scripts which have been developed in order to run and test the application. This includes scripts for capturing images via a webcam, gathering data regarding the objects found in the image, and pushing this data to the Firebase database which has been implemented for storage. Due to the fact that the current object recognition script is not fully functional, also included is a proof of concept CRUD file, which is capable of successfully pushing random values to the database rather than inaccurate data from the object recognition script.

The code structure description has been split into segments by file, and further segmented by the functionality of the code. Included will be detailed segments of the code with accompanying description which will explain the functionality of that segment.

30 Seconds

Import Statements

The import statement for cv2 is used in order to connect to the image capturing library in order to access and use the camera. The import statement for numpy as np is used in order to connect to the numpy library which is used for scientific computing when storing large multi-dimensional arrays, which are used when storing object recognition data. The import statement for time is used to access the system time, and calculate when the recording that will be taken is stopped.

Creating VideoCapture Object

The value cap uses the cv2 import in order to create a VideoCapture object, with a parameter of 0 to represent the default camera that the system detects. We then set the integer value of capture_duration to 20 seconds, which we will use in order to set the amount of time that the video is being captured for. We then check to ensure that the camera has been opened successfully, and if it is not we display a message stating that the camera feed was unable to be opened successfully.

The values for frame_width and frame_height are assigned the values of cap.get(3) and cap.get(4) respectively. These values are used to set the resolutions of the frame, as the default resolutions are system dependent. We also compute these resolutions from floating point numbers to integer values.

Create VideoWriter Object

The variable out is used in order to create and store a VideoWriter object, which will store the output in the parameter ‘output.avi’. The parameter cv2.VideoWriter_fourcc(‘M’, ‘J’, ‘P’, ‘G’) sets the encoding, in other terms what kind of video file is being captured. The parameter 20 dictates the number of frames per second. The parameter (frame_width, frame_height) dictates the size of the camera frame.

Video Capturing

We use the variable start_time in order to determine the time at which the video capture begins to take place. We then create a while loop, which will continue until an integer value equaling the difference between the current time and the value stored in start_time is greater than the value stored in capture_duration (as mentioned previously, this value is equal 20).

The following occurs within each iteration of the while loop, until the capture_duration has been surpassed. The values ret and frame are both being assigned to the value cap.read(), which does not include parameters. The variable ret will obtain a Boolean value from the camera frame that is either True or False, depending on whether the camera feed is being retrieved successfully. The variable frame will obtain the next image from the camera.

If ret returns a False value, then the while loop is broken, as the script is no longer able to successfully read frames from the camera. If ret returns a True value, then the variable frame is written to the output file using out.write(frame).

In the case that the video feed is being displayed on a monitor and a user wishes to dictate when the camera should stop recording, the following code is also in place. The value cv2.imshow('frame', frame) is used to display the current frame on the monitor. The if statement is used to give the user a distinct way to stop the script from running. This can be accomplished by pressing the value q on the keyboard.

Release and Close Frames

Once the video has been properly captured, both the cap and out variables may be released. This stops the VideoCapture and VideoWrite objects from continuing. We then use cv2.destroyAllWindows() in order to destroy the frames which have been used.

Photo Capturing

Imports

The import statement for cv2 is used in order to connect to the image capturing library in order to access and use the camera device.

Constructor

The constructor is used in order to set the amount of time between each image capture, this will be determined by the runner class when the webcam object is created and initialized. In the case that another value is passed into the constructor, this value will replace the default value of 5.

Image Capture

This method is used to capture the photograph of the parking lot. This method is used in the runner class, which is tracking the amount of time between each image capture.

A try statement is included in order to determine whether the image capture was successful. It may not be based on the availability of a webcam. Within the statement, the variable check and frame read in the image, with check retrieving information regarding whether it was able to retrieve information from the camera and frame retrieving the matrix with the image. The variable img_name is included to determine the name of the file which the captured image should be printed to, and the method cv2.imwrite() writes the image to said file.

The methods release() and destroyAllWindows() are used for closing the webcam so that it can be used again to capture another photograph. The name of the image is then returned from the method.

Single Image Counting

Imports

We import TensorFlow, a machine learning library which is used within the API that we have selected in order to properly perform image recognition.

Object Detection Imports

The import for backbone specifies the object detection model that we will be using throughout the code. The object_counting_api is used to actually perform the counting that needs to take place in order to report back the number of cars which are currently in the parking lot.

GitHub Connection

We gather the image which we will be passing to the object detection API, and pass it in to object_counting_api to gather the number of objects recognized. The variable detection_graph is assigned the value of what level of accuracy we are able to achieve for each object detected. The category_index is a list of the objects that we are able to accurately detect.

The variable is_color_recognition is used to determine whether the image captured is a color image or black and white. The variable result again uses the object_counting_api to determine the objects that were found, and the number of these objects that have been detected. This result is then printed and returned so that the runner method is able to split the string to determine the number of cars present in the parking lot.

Runner

Imports

The import time is used in order to track the amount of time between image captures. The import Webcam from class Vision is used in order to capture images and create instances of the Webcam class. The import single_image_object_counting is used in order to pass the captured photographs into the object detection model.

Image Capture and Processing

The variable cam1 is used in order to set the image capture time interval to a total of 5 seconds. The variable start_time is then used in order to store the current time from the system. Within the while loop, there is a check to determine whether the interval between the time that the application was started and the current time is a great enough difference to warrant the capture of an additional image. Once this criteria is met, the start_time is then reset to the new system time.

The method capture from the Webcam class is used in order to capture and store a Webcam image within the variable image. This image is then used as a parameter in the count() method, in order to gather the result of the number and type of objects that occur in the image is stored in a string within the currCount variable. The variable currCount is then used in order to split the string and locate the value associated with the number of cars that are currently in the parking lot. If this information occurs within the string, then the value is then printed to the console, if this information does not appear within the string, the string value “no cars found” is printed instead.

Database Pushes

Import Statements

The import statement firebase is used in order to connect the script to our Firebase database, which currently stores data regarding the number of spots currently taken and the number of spots which are available in the parking lot. The import statement for seed is used in order to constantly create new seeds based on the current time, adding an extra layer of randomness to the numbers being pushed to the database. The import statement for time is used in order to generate pseudo random numbers in order to push them to the database. The import statement for randint is used in order to include the Random class, allowing developers to create and use random numbers as necessary.

Connection with Firebase

Creates a Firebase object which will allow the data to be manipulated from the script. The object takes as input the link to the CWU Parking Application Firebase database. The input None is used for authentication, since there is no authentication specified within the database.

Pushing and Printing Random Value

The method seed(time.time()) is used in order to create a starting point in the generation of random numbers. A random seed is able to specify the start point at which a computer generates a random number sequence. This can be any number, but it typically comes from the seconds on the associated computer system's clock.

The variable res is used to store a random integer from 1 to 60 using the method randint. The variable res is then used when using the .put() function, placing the data in the correct field in Firebase. The parameters for this function are as follows: ' ', 'parkingLots/lot1/SPOTS_TAKEN', and res. The empty parameter ADD DETAIL HERE. The 'parkingLots/lot1/SPOTS_TAKEN parameter' dictates the point in the database where the value will be stored. The parameter res is the value which is being placed within the database.

This function will place the information into the database, and an accompanying print statement will print the value that has been pushed to the console.

User Interface Design

The following code structure description has been formatted such that the code is divided first by the project folder in which the file is saved, and then by the functionality of the code. Included will be specific segments of the code with accompanying description of what that segment of the code is doing for the project. From this information, the reader should be able to understand in detail how the code is working to create and run the application.

Assets

The assets folder includes information regarding the fonts and photos that are used throughout the various screens in the application.

Components

HeaderButton.js

The following is a description of the JavaScript code which is used to create easily accessed header button options across various screens in the application.

The code uses the import React in order to install and use React libraries. The import for Platform is used in order to determine the operating system that the device accessing the application is using, which allows the application to make adjustments accordingly. The import for HeaderButton allows us to manipulate and choose which button should be included in the header. The import for Ionicons is to access the list of different button icons which may be used in order to include images within the header.

The code itself sets the initial size and color of the Ionicon. This can be rewritten later in other areas of the code, but this creates a baseline for all the pages to use.

Fire.js

The following is a description of the JavaScript code which is used to access the Firebase database. Within the database we have stored constant integers which represent the total number of parking spots which are available in the parking lot. There is also an integer value which stores the number of parking spots which are currently being occupied within the parking lot.

An import firebase is used in order to access the information within the database. The variable firebaseConfig is populated with data that was provided via Firebase and copied into the file. We then create a variable fire which we use to initialize the database using the information in firebaseConfig. The variable fire is then exported, so that the firebase may be accessed via that variable throughout the application.

Constants

Colors.js

This file is used in order to export several variables so that they may be used as constants throughout the application, rather than having to track the color values throughout the process. The colors which we have chosen to include here are cwuRed and cwuBlack, which we retrieved from the Central Washington University website.

Data

Parking-Lot-Data.json

This file stores information regarding the parking spots which we have currently gathered data for. We have opted to store information which is used frequently and which does not require updating here rather than on the database , in order to access the information quickly and efficiently.

Navigation

ParkingAppNavigator.js

The import statement for React allows us to use all of the React libraries. The import for createAppContainer creates an app container which can then be exported. The import for createStackNavigator is used in order to store information regarding the navigation, and push and pull this information as needed.

The import statements for LandingScreen, MapScreen, and SettingsScreen are used in order to access the various pages of the application quickly and efficiently. The import statement for Colors allows the developer access to the preset colors that have been previously discussed, cwuRed and cwuBlack.

The code which is included allows navigation between the pages through access to this file and selection of the various switch objects.

Screens

Landing Screen

Imports

The import statement for React allows us to use all of the React libraries. The import for View is used to determine how the screen visually appears to the user. This is further defined within the stylesheet, which is used to set the exact parameters for View in an easier format. The import for Text is used to display text to the user on the screen throughout this screen of the application. The import for StyleSheet is used so that the developer may create a list containing the different types of styles that are used on the page. The import for ImageBackground is used in order to include an image in the background of the screen. In this case, the image which is included is the Central Washington University logo. The import for TouchableOpacity is being used to determine the style of the button which appears on the page and what happens when a user presses that button. The import for Colors is used to access a local file, which stores values for the variables cwuRed and cwuBlack.

Screen Display

This code is used to access the Stylesheet options to display pertinent information to the user. For the most part, there are no methods being used throughout the code, as the screen only has one function.

The onPress={() => {props.navigation.navigate("Map");}} is a method which is used to determine the action which takes place once the user selects the button which reads 'I Agree'. When the user selects this option, props.naviation.navigate("Map") is used to navigate the user to the accompanying Maps screen in order to continue their user process.

Navigation Options

This code sets the navigation options within the ParkingAppNagivator.js file which was previously discussed. This is used to display the header for the Landing Screen of the application, specifically, the text "Legal Agreement".

Style Sheet

This sets the code for the stylesheet which is being accessed throughout the screen display on Landing Screen. This includes text and background settings, as well as the dimensions of the screen.

Maps Screen

Imports

The import for React is used for accessing the React JavaScript libraries. The import useState is used for keeping track of a state variable. The import for useEffect is used so that whenever an action occurs, the screen immediately updates all information.

The import for View is used to determine how the screen will appear for the user. The import for Text is used in order to display textual information to the screen for the user. The import for Stylesheet is used in order to access the stylesheet for the page, which contains information regarding the visual design of the screen and it's dimensions. The import for Platform is used in order to determine whether the user is using the Android or iOS operating system; this is important information in the few cases that code needs to be modified in order to be displayed properly on both platforms. The import for Dimensions is used in order to determine the current dimensions of the screen, which will need to be accessed in order to dictate the width of the map which is to be displayed. The import for TouchableOpacity is used

in order to create aesthetically pleasing and functional buttons. The import for Switch is used in order to create a toggle to determine the user's mode of transportation.

The imports HeaderButtons and HeaderButton work concurrently to allow the settings icon to appear in the upper right hand corner of the screen. The imports for MapView are used in order to determine how the map appears to the user. This includes imports for the elements Marker for setting the location of parking lot pins, Callout to display information to the user based on the parking lot which they have selected, Polygon in order to draw the actual shape of the parking lot on the screen, and PROVIDER_GOOGLE in order to verify that the Google API is being used in order to display the map and directions.

The import for MapViewDirections is used in order to allow the directions line to be displayed to the user. The import for fire is used in order to access information from the Firebase database, which currently stores information regarding the number of total parking spots which are available in each registered parking lot, and the number of parking spots in each parking lot which are currently occupied. The import for Toast is used in order to allow the user of the Redirection wheel, which shows the user a spinning wheel signifying that the user's redirection option is being calculated. The import for Colors is used in order to access the constants which have been created for the colors cwuRed and cwuBlack. The import for parkingLotData gains access to the JSON data file which contains constant information about each parking lot.

Map Settings

In order to initially set up the maps display, the variable dHeight is used in order to store the dimensions of the current window of the screen. This means that it uses the dimensions of the phone on which the application is running. The variables LATITUDE_DELTA and LONGITUDE_DELTA are used in order to determine the initial region of the map which will be displayed to the user when the application is opened. The user's destination is gathered using watchOptions, with an accuracy of within 5 meters of the user's actual location. The const for intialMapRegion uses the latitude and longitude in order to set the initial map region.

Firebase Querying

The method fetchLotData is used in order to gather information from the Firebase database regarding each parking lot. This method fills an array numSpotsFreeArr with the number of parking spots available in each lot, using a calculation which finds the difference between the queried values for the total number of spots and the number of spots occupied. This array will be accessed in other areas of the code in order to display the information to users, as well as using the calculation in order to determine whether the user needs to be redirected to another parking lot due to the fact that theirs has filled.

Fetching User Location

The method getUserLocation is used in order to access the geolocation of the user from their mobile device. The user will be prompted to give the application access to this information prior to being allowed to fully access the maps screen. This function uses another function, geolocation.getCurrentPosition() in order to gather information regarding the current latitude and longitude of the user's location. The method setUserOrigin is then used in order to set the user's initial location pin.

Updating User Destination

The method toggleUserDest takes as input the parking lot which the user has selected, or the parking lot which they are to be redirected to. This is achieved by returning latitude and longitude values associated with the pin which represents the parking lot which has been selected.

Transportation Toggle

The function toggleMode() is used with the parameter value, which represents the current mode of transportation. The values which are currently accepted regarding modes of transportation are walking and driving. The variable value is then passed to the setTrueFalseUM method in order to swap the current mode of transportation with the opposite value in the variable trueFalseUM. Once this has been accomplished, the method checks the variable trueFalseUM in order to determine whether the parameter for pre-built method setUserMode is set to the value “DRIVING” or “WALKING”.

Redirection Function

The function redirection() is used in order to access information regarding the parking lot that is closest to the user’s parking lot of choice, in the case that the parking lot which the user originally selected has filled. The function accomplishes this by creating an object lotData of type parkingLotData. This object is then able to access information regarding the closest parking lot to the user’s selected parking lot, by checking the LOT_CLOSEST associated with that lotIndex within the JSON data file described previously. The value for lotData[lotIndex].LOT_CLOSEST is assigned to the variable newLotID.

A for loop is then used in order to go through the JSON data file to find the index of lotData which has a LOT_ID which matches the value of newLotData. Once this index has been located, the variable newDest is assigned the value of toggleUserDest(lotData[i]), a function which will return the coordinate values of the new parking lot. setLotIndex is then used in order to set the lotIndex to the new value, and the value of newDest is returned.

Parking Lot Filled Check

The useEffect function constantly checks whether the parking lot the user originally selected has filled since the user requested directions. The function does this by checking the value of the numSpotsFreeArr at the current lotIndex. When the value reaches zero, the user will be informed that they are being redirected via a brief popup which disappears after two seconds. The variable actualUserDest is then set to the value of the function redirection() which will return the coordinates of the parking lot that the user is now being directed to. This value is then used to update userDest using setUserDest. The user’s new location is then fetched, as well as new parking lot data via the getUserDirection and fetchLotData functions, respectively.

Map Display

In order to determine how the Google Maps API will appear on the screen, values for MapView are set according to the preferences of developers. In the case that future versions of the CWUParkingApplication would like to change the current set up, these variables can be changed accordingly.

Polygon Rendering

The code which is used in order to outline and fill the polygons representing the shapes of the actual parking lots first checks that the parking lot should be shown. Parking lots should currently be shown at all times, even when filters dictate that the pins representing parking lot selection should be hidden. The

Polygon element is then given information regarding the LOT_ID of each parking lot, as well as the coordinates which would need to be drawn.

Setting Lot Index

Prior to setting a lot, in order to display the included parking lots the function map is used. The function navigates the array parkingLots, which is created from the local JSON file containing constant data for each parking lot. The application uses the coordinates associated with each parking lot, which are saved within this data file in order to map the polygon and pin coordinates.

The code which is used in order to dictate which pins should be displayed for parking lot selection is constantly tracking information regarding whether the parking lot should be shown according to the filters which the user has set. The code does this using the tracksInfoWindowChanges method. The pin is then placed using the values for lot.PIN_COORDINATES[0] and lot.PIN_COORDINATES[1], which are assigned to the values of latitude and longitude respectively. The pinColor is based on the value of LOT_COLOR, which is listed in the JSON data file discussed previously.

The onPress method which is used within this code detects whether the user selects one of the pins representing the parking lots on the screen. In the event that the user selects one of these pins, the value of selectedLot is defined using the method setSelectedLot(lot). Based on the LOT_ID of said lot, the lotIndex is then set using the setLotIndex method, and the variable actualUserDest is set to toggleUserDest(lot), to be used in order to direct the user to their final destination.

Callout

The Callout element uses the tooltip onPress in order to display the user information about a parking lot once they have selected the pin associated with that parking lot on the map. This onPress method sets the userDest using the setUserDest() method with actualUserDest as a parameter.

Directions Line Display

The MapViewDirections element is used in order to set user preferences regarding how the user should be displayed directions to their final location. In this case the main information that is set is the color of the directions line, and information regarding what mode of transportation will be used to arrive at the final destination. For the current version of the application, the only modes of transportation currently available are “WALKING” and “DRIVING”.

“Walking” vs “Driving”

The Switch element is used in order to create a switch which will determine whether the user is “DRIVING” or “WALKING” to their final destination. The variable value is set to the current value of the constant trueFalseUM. Once this information is changed, the newValue is given as parameter to the method toggleMode(), which will switch the user’s current mode of transportation to the opposite, resetting the directions line to also signify this change if need be.

Clear Directions

The TouchableOpacity element is used in order to create a button which provides the user the option to clear the current directions from the screen. When this button is selected, the onPress() method is used in order to use the setUserDest to the userOrigin, which is the user’s current location. This effectively clears the directions from the screen.

Navigation

The method for MapScreen.navigationOptions is used to include the header at the top of the screen. This is necessary in order to use the element HeaderButtons, which is used in order to include the settings icon in the upper right hand corner of the screen. This settings button is used in order to transfer the user from the current screen to the Settings screen, where they will be able to apply the filters they require.

Style Sheet

This sets the code for the stylesheet which is being accessed throughout the screen display on Maps Screen. This includes text and background settings, as well as the dimensions of the screen.

Settings Screen

Imports

The import for useState is used for keeping a state track of a variable. The import for useEffect is used so that whenever an action occurs, the screen immediately updates all information. The import for View is used to determine how the screen will appear to the user. The import for Text allows text to be displayed to the user on the screen. The import Switch is used for the creation of the switches that allow the filters to be applied. The import for Platform is used to determine whether the application is being run in iOS or Android, as this will determine the thumb track color of the switch.

The imports for HeaderButton and HeaderButtons work side by side in order to allow the back button in the upper right corner to appear correctly on the screen. The import for parkingLotData is used in order to access the data for the parking lots which is stored locally, as discussed previously. We also import Colors, in order to access the colors cwuRed and cwuBlack.

Filters

We begin by initializing all markers using the lotData variable, which is assigned the array parkingLots from parkingLotData. We then use a for loop in order to go through this array to check the lot type, and based on these values sets whether the lots will be shown initially. The const values are then to set the state of switches based on this information. The method useEffect() receives information regarding which lots to show, based on the switches that have been activated. This information can then be pulled from the Maps screen in order to determine which of the lots should be displayed.

Display

This code is used to access the Stylesheet options to display pertinent information and handle movements of the switches according to the user's touch input. The method that is used to perform this switch is onValueChange(). onValueChange() registers whether a switch has been touched by the user, and toggles the switch. It also changes the variable for that switch to the opposite value, so that it can be registered throughout the code that the switch has been activated.

Navigation

This code is included in order to allow the user to navigate from the Settings screen back to the Maps screen, where the changes that they have made should have been implemented. This is done using the HeaderButtons import previously mentioned.

Style Sheet

This sets the code for the stylesheet which is being accessed throughout the screen display on Settings Screen. This includes text and background settings, as well as the dimensions of the screen.

App.js

This file is the default file that is needed in order to run the program. It is the initial file that is searched for when the 'npm start' command is used in order to run the application. The React import is used in order to give developers access to all of the React libraries. The import for ParkingAppNavigator is used in order to access the pages which have been designed for the application. We are able to navigate the user to the initial screen with the export statement listed, which access the first page listed within the PakingAppNavigator.js file discussed previously.

App.json

This file contains information regarding the operating systems which are supported on the application. For iOS, this means setting that tablet devices may also be supported when using the application. For Android, this means setting that the user's location must be accessible in order to provide accurate directions, as well as update the user's location on the map as they are travelling to their destination.

Package-lock.json

This file is automatically generated upon creation of the project, and contains information regarding the imports that the application supports initially. This file is added to automatically, depending on the additional libraries that are installed throughout the creation of the application.

Package.json

This file is automatically generated and contains initial commands which can be used when creating the application, as well as the initial key dependencies.

Testing

Purpose of Testing

The purpose of testing in the CWU Parking Application is to ensure that users are getting accurate data regarding the parking lots on the Ellensburg campus. This data includes the number of parking spots available in each parking lot and directions to such parking lots.

Types of Testing

The development team was able to concurrently test the Android and iOS platforms of the application using the Expo application for Android and iOS devices. The type of testing which was used for the project is white box testing, as those who were involved in the testing process were also involved in the development of the application.

Testing was performed regarding the accurate querying of the number of currently occupied parking spaces in each parking lot, by manually placing random values into the database in order to ensure certain conditions. These conditions included both the increase and decrease of the number of cars which are currently occupying the parking lot. The development team was able to successfully query such information at a constant rate.

A crucial condition that was also tested for is the case that the parking lot that the user selected fills prior to their arrival at their destination. Should this case occur, the application should inform the user of this issue and automatically redirect them to the nearest parking lot possible. This condition was tested for again using manual input to the database, while the tester in question was in transit to their destination.

Project Limitations

The main limitations associated with the project involved the object recognition that was intended to track the number of parking spots which are occupied in real-time. While the development team was able to implement an API with some success, but were unable to increase the accuracy of the API used for our own model.

Conclusion

The purpose of this document is to identify and explain in detail the inner workings of the initial version of the CWU Parking Application. This document may be referred to as a blueprint, should a developer be interested in continuing to work on or completely recreate the application. In the future, the development team aims to further refine the object recognition script, so that we may be able to track the current number of cars within a parking lot with greater accuracy. If there are any questions or concerns regarding any of the information outlined above, the development team may be reached through Hailey Lawton at lawtonh@cwu.edu or Kevin Bertelsen at bertelsenk@cwu.edu.