Swift Park
An Application for the Internet of Things

Brien Elwood Washburn, Kohl Riekhof, Jia Jun Yu
University of Utah

Abstract— **The internet is currently on the doorstep of the next stage in its evolution—the internet of things. The internet of things is a vision of the future in which things in the world can talk to each other. A current problem that can be addressed by implementing the internet of things is parking, particularly at universities. This report details the development of Swift Park, which is an application of the internet of things. Swift Park is an application that looks to a future where smart cars with a plethora of cameras for self-driving or monitoring the road will be common place. Swift Park uses cameras in smart cars to capture QR codes placed on posts in each stall of a parking lot. The data in the QR code is relayed to the URL found by processing the QR code using custom software in the car. This data is received by a controller developed in Python that places it in our Firebase database. The Swift Park website, developed in CSS, JQuery, JavaScript, and HTML5, listens asynchronously to the database and updates a map, which anyone can access at the Swift Park website, in real time.**

I. Introduction

The internet of things is a vision of the future where objects throughout the world will contain and transmit data. As the name implies, common appliances and structures in our environment will be capable of connecting to the internet in a meaningful way. The goal of this vision is to connect the ordinary objects in life to eliminate the mundane, such as shopping for food or watering plants. The internet of things is a vision of productivity.

Parking, especially at universities which sell more passes than spaces, can be time-consuming. Lots are often far apart, or have special rules about which passes can be used during certain times of the day. It is inconvenient to drive each row of the lot only to realize that there aren't any spaces left. Navigating multiple lots in search of a place to park can take up to 10 minutes.

Sometimes the best option is to park in a paid stall when there is no time to wander the lot looking for a spot. Current systems require the user to be carrying change or manually enter information into an app each time they park in a paid space. While paying with an app is a simple process, it suffers from the same problems that traditional meters have—it is necessary to correctly estimate how long you will need to park. Oftentimes, these services require a surcharge for processing the payment, making it even more costly to extend the time on a paid stall.

Both of these problems can be mitigated with the Swift Park lot-monitoring system in a future where smart cars are ubiquitous. (For the purpose of this report, smart car(s) will refer to any Wi-Fi capable car or cars that have cameras in the front and back of the car capable of scanning a QR code and providing it to custom software in the system.) The Swift Park system involves placing QR codes on a post in each parking space in the lot. Each person that frequents the lot will have a smart car that will constantly be scanning its surroundings. When the car pulls into a parking stall, the cameras will capture the QR code and pass it to the Swift Park software suite. The QR code will be evaluated and the Swift Park software will send the data to the URL found in the QR code. The server at the URL will receive the data, update the parking-lot database, and the Swift Park website will then update the map of the parking lot to acknowledge the new occupant.

Swift Park is a vision of the future. In this future, every car will come straight from the manufacturer with hardware that can support QR capture and communication through the internet-of-things (IoT) protocol defined in the software of the car's computer. This IoT application allows for easier parking, and will increase revenue of paid parking given that users are held accountable for every second they are parked in the stall. It can also decrease costs as lots won't need to be monitored as heavily given the ability to automatically recognize a car and have the user pay for their time spent parking. Additionally, parking passes

can be validated through the Swift Park database.

The remainder of the report discusses the evolution of the internet and similar applications in Background; Results

II. Background

A. Evolution of the Internet

The internet began as a way to connect computers to other computers. It was created in universities as a tool of academia. With the advent of the personal computer, the internet became a tool for people to connect with other people. Social networks see widespread use, as well as sites used for business and education.

As technology has improved over the last decade, the size of the hardware has become drastically smaller and more powerful. The internet has become common in nearly every aspect of our lives, and is now used to keep in constant contact with people through messaging and email.

Power is now so common throughout our world that devices can be placed nearly anywhere. Now that computers can be embedded into just about anything, the internet is going through another stage in its evolution: one in which objects can communicate with one another in the hopes of greater efficiency with less human effort. This next step in the evolution of the internet is the internet of things.

B. Internet of Things Application

A study was conducted about the use of smart objects as the building blocks for the internet of things [1]. This study involved designing smart objects with different design categories such as awareness and interactions. The three types of objects were activity-aware objects, policy-aware objects, and process-aware objects.

Activity-aware objects are capable of recording information about how the tool was used. This data includes metrics such as time, state, and vibration of the object.

Policy-aware objects are activity-aware objects that are capable of evaluating the data stored from activity-aware objects and taking certain action based upon that data.

One example used was a smart barrel that was capable of informing workers in the vicinity of the barrel whether the current state of the chemicals within the barrel posed a risk to workers based on its current condition.

This is similar to the paid-parking functionality in Swift Park that is capable of taking certain actions based upon the interactions between a passive and active IoT component. If a person is in paid parking, the system is able to use information from the passive and active IoT components to charge the user's credit card according to the time that they have been parked.

C. Lot Monitoring

Smart Lot is a senior project completed at the University of Utah in 2010. The Smart Lot system uses a camera to locate empty parking spaces and then relays that information to traffic lights stationed at the end of each row. The challenges this system faces stem primarily from the parking-space detection algorithm. It encounters issues with different lighting, snow, and potentially from the color of cars.

Swarco Traffic Systems is a lot-monitoring system that uses ultrasonic technology to determine where cars are parked [2]. The ultrasonic sensors are capable of measuring the distance of an object to the ground. If the distance traveled by the wave is determined to be reduced by the presence of car, the sensor sends an "Occupied" status message to a higher-level area controller. These controllers then communicate with hardware that displays information about parking on signage inside of the parking lot.

Streetline uses a mesh network to relay information about free parking spaces and volume and speed of passing traffic [3]. Each node in the network has a magnetic sensor that detects cars parked above it. The sensor is capable of determining if a car is above it by searching for disturbances in the earth's magnetic field. This information is then transmitted wirelessly through the mesh network. A mesh network allows for each mesh node to collect and transfer information through other nodes and eventually to servers that can display free parking spots through street signage or smartphone maps.

The Swift Park system circumvents the detection problems encountered by Smart Lot by providing each spot with its own QR code. This will allow accurate sensing of individual parking spaces despite environmental factors such as weather or the time of day.

The first primary difference between Swarco Traffic Systems, Streetline, and Swift Park is the cost. The sensors for these two systems are more complex due to their sensing style and the ability to communicate wirelessly. The QR codes used in the Swift Park system can be made at a fraction of the cost, and there is no additional cost sensing the QR codes since the application is targeted for a time when smart cars

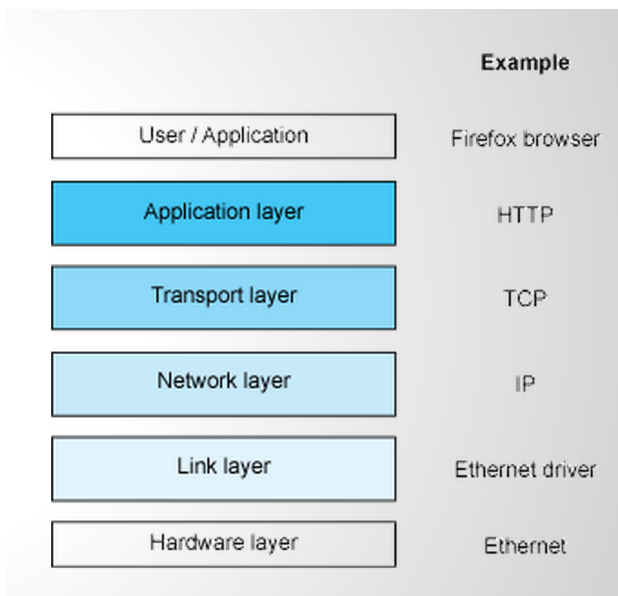

Fig. 1. The common five-layer protocol stack used in networking.

are common, which leaves only the software to be downloaded to process the QR code and send it to the controller over Wi-Fi. Swift Park also utilizes a dedicated website to display lot information instead of hardware in the lot or existing maps for smartphones.

The second major difference is the route to communicate information and the way sensing works. These others systems use intelligent sensors that are separate from the car to locate cars and transmit data. This means that once a sensor is put in place, it has no reliance on the user's system to operate correctly, making it harder to tamper with. Swift Park moves the active components to the user's car, thus making the sensors very simple, cheap, and easy to install and maintain, but there is reliance on the user's system to make Swift Park work correctly.

III. Results

As computer engineers, it is important to understand current technology while having an eye on the future. Computer engineering is an innovative and competitive field, and it is an important skill to be able see the trends in technology.

The central idea of Swift Park was to create a project that utilizes current technologies while predicting and making full use of upcoming, revolutionary technologies: namely, smart cars and the internet of things.

The internet of things is going to allow nearly any object to provide data to the world around it through two component types: active components and passive components. Passive items—things like a parking stall, a sign, or a product in a grocery store—will be capable of providing information to the environment. Active items, such as RFID readers or cameras linked with QR-processing technology, will be able to scan the environment and access the information in passive items. The active component, after acquiring information from a passive component, will hand off the data to a client. The client can then process the data according to its established IoT protocol, completing the data transfer from a passive object all the way up to software that can process and use the information encoded in the passive item.

One possible application is a grocery store that marks each item in the store with an RFID tag. Readers at the exits of the store would scan each item and send the information to varying databases. One of the databases could group items by user to allow the grocery store to determine which items are typically purchased together. This information could then be used to optimize the layout of items in the grocery store to make it easier for customers to find items and maximize profits.

In Swift Park, the passive object is a parking stall that contains a QR code. The active component, also called the client, is a smart car with cameras, QR-processing software, Wi-Fi capabilities, and the Swift Park application suite. The client engages with the passive parking spot when the car pulls into a stall. The client then sends the data it receives from the parking stall's QR code to the URL packaged in a JavaScript Object Notation (JSON) object. The URL maps to a controller that parses the JSON object and acts according to the data acquired. If the car is parking in the space it occupies, the client updates that parking space's entry in the Firebase database to reflect the change in the parking lot. The Swift Park website then updates its map of the parking lot asynchronously based on the changes to the database.

A. RFID to QR Technology

The project proposal originally stated that the passive IoT component in Swift Park was to be an unpowered ultra-high-frequency radio-frequency identification (UHF RFID) tag. An unpowered UHF RFID tag is called a passive tag because it does not require any active component to function. The tag is accessed using a UHF RFID reader that can access the information.

The plan was to place the passive UHF RFID tag an inch or two deep into the concrete of each parking stall. A UHF RFID reader, the active IoT component,

would then be connected to a Raspberry Pi computer inside the car that could read activate the reader when the car stopped. The reader would then power the UHF RFID tag in the concrete and retrieve the information it was storing.

There were two problems that came up. The first was that the distance from the reader inside the car to the tag buried in the concrete is in the mid range of the limit of UHF RFID. The second is that the obstructions of nearby metals (the cars in the lot) and concrete significantly reduce the range. These two problems together make it impossible to read the UHF RFID tags from the car, even if the reader is outside and on the bottom of the car.

This was discovered when attempting to validate the plan to use RFID. Contact was made with a representative of SkyRFID to learn more about what tags and readers in particular would work well with the project. The response from an employee of SkyRFID named Dylan made it clear that "UHF 433MHz active-tag technology" would be required to read the tag with the car and concrete as obstructions and at the distance we needed.

It became necessary to find another technology that would be cheap, passive, and would fit with the internet-of-things core of the project. QR was the next technology that made sense to try. QR codes are becoming a popular choice to connect items to the internet of things because they are cheap, easy to generate (there are a number of websites that will generate QR codes for free), and can be very small and unobtrusive. They are also capable of storing up to 3kb of data.

QR codes were a great choice for Swift Park as it is a passive technology that makes it simple and cheap to connect a lot of passive components to the system, allowing the less numerous active components to do the heavy lifting. They are also able to hold as much information as we needed without being too small so they are able to scan and process quickly.

B. Research and Protocol Definition

The research portion of the project was centered on defining two protocols to communicate with active and passive IoT hardware. The passive IoT protocol is used to communicate with hardware that cannot process information. When two active IoT items such as a server and Swift Park hub are communicating, the active IoT protocol is used.

When innovating it is unnecessary to reinvent the wheel, so we made use of of the current protocol stack utilized in networking. The protocol stack consists of five layers. From bottom to top, these layers are: physical (or hardware), link, network, transport, and application layers. The protocol stack can be seen in Fig. 1, with common protocols to the right of the stack. (The User/Application layer is not a part of the five-layer protocol stack—it is used for the express purpose of showing a typical network from top to bottom.)

The physical layer describes the medium over which the information travels. In this case, information will travel on electromagnetic waves.

The link layer details the protocol that controls the physical layer. In the Swift Park project, the link layer supports two protocols: passive IoT using the using cameras to read QR codes, and Wi-Fi (IEEE 802.11), which will communicate information to the Python server (the controller).

The network and transport will maintain their nearly shared functionality of making certain that messages are accurately passed from the user, down through the protocol stack, and back up to web page.

The application layer is traditionally used for the HTTP protocol for requesting information about web pages on the internet. In Swift Park, the application layer is used to facilitate communication over the active IoT protocol, which uses HTTP as the base protocol.

When communicating via QR code, it is necessary to minimize the complexity of the protocol. This means that the QR codes need to hold a small amount of information, namely only a uniform resource locator (URL) and a value, so that the image can be captured and analyzed rapidly. The data is then sent to the URL from the QR code. The controller at the URL can then interact with the database or provide information back to the client.

The active IoT protocol is then used between active IoT components which are able to process information. The active protocol is capable of communicating a variety of actions which are specified in the handshake between two active components. Each active IoT component can then provide requests to the other IoT component, as well as the parameters required to enact the request.

1) Passive IoT Protocol: The passive IoT protocol is for active components to talk to passive components. The passive protocol is very simple so that the passive component does not require a lot of resources and can communicate rapidly.

The passive IoT protocol for Swift Park contains only two pieces of information: the data to be sent, and the

URL to which it will be sent. The format of the QR code is

passive IoT: data+host

An example QR code from Swift Park that is used for one of the parking spaces is

passive IoT: 5+lab1-18.eng.utah.edu

This information will allow the client to send the value 5, which represents the parking-stall number 5, to the URL lab1-18.eng.utah.edu. This will be done using the active IoT protocol.

2) Active IoT Protocol: The active IoT protocol is used when two active components are talking to one another. In the case of Swift Park, the two active components are the client and the controller. The active IoT protocol defined by Swift Park uses a JSON object to pass data between two active IoT objects.

JSON, or JavaScript Object Notation, is a way to format data to make it easy to exchange and understand. It is language-independent, which makes it simple to use across any language or platform, and it is ”self-describing”, meaning it makes intuitive sense what the data means when looking at it. As can be inferred from the name, it uses JavaScript formatting, but it is only text.

The base protocol used to communicate between the client and controller in active IoT is the HyperText Transfer Protocol (HTTP). HTTP is foundational to the internet, and is the most common application-layer protocol used online.

The active IoT protocol defined for Swift Park is strictly a data-formatting protocol, and exists in two different forms. One is for when the client talks to the controller, and the other is for the controller to the client.

The client-to-controller protocol states that a JSON object will be passed over HTTP in the format of:

active IoT:
data={’type’:REQUEST,’spot’:SPOT,’user’:USER}

where REQUEST is the type request being made of the controller (e.g., claim spot or release spot), SPOT is the parking stall that the user is occupying, and USER is the unique identification of the client.

The active IoT protocol for the controller to talk to the user also uses a JSON object over HTTP, but the format is a bit different. Instead, the JSON format with information useful to the client. The format is

active IoT: data={’available’:AVAILABLE,
’permission’:PERMISSION,’payed’:PAYED}

where AVAILABLE is set based on whether the parking stall is available to park in, PERMISSION tells the client if they are allowed to park in the spot (parking in a handicap spot without handicap pass would result in denial of permission), and PAYED states whether parking in the space occupied requires payment.

C. The Client

The client can be thought of as the system that interacts with the lot and conveys information to the controller. The client is the system composed of the cameras in the smart car, the software that evaluates the QR images it receives, and the software and hardware that relays the parking-spot information to the controller. The client is coded entirely in Python 2.7, and uses OpenCV and ZBar for image processing.

The client works by constantly taking images and evaluating them to see if they are a QR code. If the image is not a QR code, the output file, qrdata.txt, is left blank. In the case that the image is a QR code, the code is evaluated and the data is placed in the qrdata.txt. Checking to see if qrdata.txt is blank is the basic test to see if the image captured was a QR code.

When the qrdata.txt contains information, the client will parse the file. The data is formatted in passive-IoT form, so the client splits the information in the file on ”+” to separate and retrieve the spot data and URL. The client then opens a socket using the Python socket library and connects to the controller server on port 2113 using the URL retrieved from the QR code. Once the client connects to the controller, it will send information to the controller using the active client-to-controller IoT protocol.

The first piece of data in the JSON object that the client sends to the controller is what action it would like to take. The second contains the parking-spot data. The third is the user’s unique Swift Park identification.

The four action options possible are request, claim, release, and close. The request action will prompt the controller to provide the client with information about the spot it currently occupies. The controller will respond using the active controller-to-client IoT protocol. The client will then parse the JSON it receives, which will tell it whether it the spot it wants to claim is available to park in, if it has permission to park in the space, and if parking the stall requires payment.

If the spot is available and the user has permission to park in the spot, the user can claim the space (details provided in the Raspberry Pi Client and Laptop Client

subsections). This requires the client to send another JSON object to the controller with the claim action. (The second and third pieces of data, spot number and user identification, respectively, will remain the same.) This will prompt the controller to claim the spot in the database, and the map of the parking lot on the website will be updated to reflect the car parked in the parking space, changing the space from green to red.

When the user is ready to leave the parking space, the client will communicate with the controller with the release action. The controller will modify the database to reflect the user leaving the lot, and the website will change the previously taken spot from red to green.

The final action possible is close. This will shut down the socket connection to the controller, ending the interaction between the client and controller.

1) Raspberry Pi Client: The original interface to simulate the smart-car client was the Raspberry Pi B+. This is a small, simple computer that has a 700MHz single-core ARM1176JZF-S CPU. (The Raspberry Pi was originally going to be used as the full client when RFID was the passive technology.) The Pi was kept when the switch to QR from RFID was made as it is a more modular element that fits the goal of the senior project well.

The Raspberry Pi has a camera called the Pi Camera that is used to capture images. The camera is accessed in the Python client through the use of an imported library called picamera. This library makes interfacing with the Pi Camera quite simple.

The Pi Camera is accessed using the command

camera = picamera.PiCamera()

This will provide a camera object that can be used to take pictures with the Pi Camera. The controller enters a loop after performing this command which will not break until the output file qrdata.txt contains QR data. Once the QR code is evaluated and its contents are placed in qrdata.txt, the file is parsed and the QR data is retrieved.

The Raspberry Pi client includes an interface comprised of LEDs and push buttons that are used to interact with the controller through the Pi's GPIO. (The interface can be seen in Fig. 2.) The orange LED on GPIO 17 is used to indicate that the client is running. The blue LED indicates that your car is currently parked and registered in the database. The RGB LED (red on GPIO 24, green on GPIO 23, and blue on GPIO 22) is used to indicate the status of the spot being parked in. If the spot is available and you have permission to park in it, the RGB LED is green. If the spot is unavailable or you don't have permission to park, the LED is red. If payment is required, the LED turns yellow.

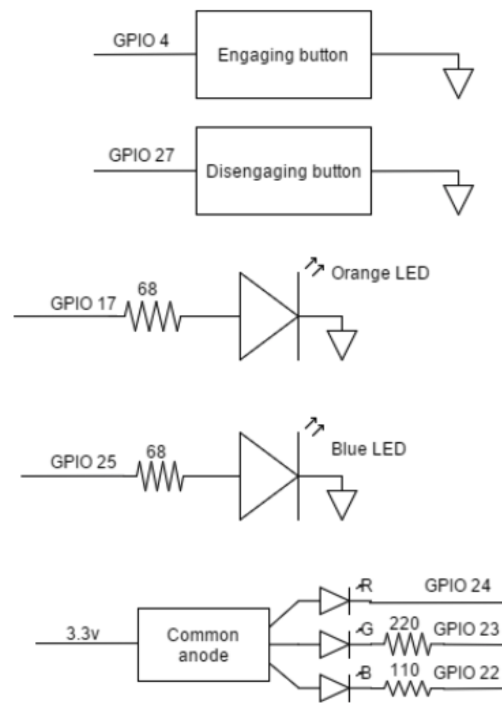

Fig. 2. The LED-and-push-button interface for the Raspberry Pi.

When the status LED is green, indicating that the user is allowed to park in the spot, the push button on GPIO 4 can be pressed. This will communicate to the controller the claim action, taking the spot. When the user is finished parking in the spot they can press the push button on GPIO 27 to send the release action to the controller, thereby freeing up the spot in the database and on the Swift Park parking-lot map.

The Raspberry Pi client with this interface is capable of being run without a display—it is entirely headless. This interface allows the user to interact with the Swift Park application without requiring any sort of display.

2) Laptop Client: The other hardware used to simulate the smart-car client is a laptop running Debian 8. The reason to use a full laptop is to get a better idea of how the system would run in an actual smart car. The laptop is able to capture and process images more rapidly, and can provide a display similar to that you would find in a modern smart car.

The laptop client is a Dell Inspiron 17 3721 with an Intel core I5-3337U processor. The processor runs at 1.8GHz, or 2.7GHz after overclocking. It also has

an Intel Centrino Ultimate-N 6300 AGN wireless card. The specifications of this machine are much closer to what could be expected from a smart car.

The laptop client captures images through the program OpenCV. OpenCV then stores the image in a folder which ZBar can access. ZBar is image-processing software, and is used to process the image to see if it is a valid QR code. If the image is a QR code, the data is extracted in placed in the qrdata.txt output file that the client can access.

The laptop client also differs from the Raspberry Pi client in that the process of interacting with the controller is automated. When the user pulls into a spot and the request action yields spot availability and permission to park from the controller, the spot is claimed automatically. Likewise, when the car leaves the spot it releases the spot automatically. The client prints information to the screen during each exchange between the client and controller so the user knows what is occurring.

D. The Database

Swift Park uses a database system called Firebase. Firebase is a back-end system based on the JavaScript library jQuery that simplifies the process of storing user information in a database, modifying the database, and extracting information from the database.

The database structure is based on the JSON format, and can be set by uploading a JSON file to Firebase. This makes it very easy to setup a database and make any necessary modifications. Firebase also has a website that makes it possible to see the database, and to view changes made to the database in real-time. The website also allows changes to be made in the database through the website GUI. This feature makes testing much easier as the JSON format is simple to understand and the real-time changes and ability to make modifications makes database manipulation incredibly quick and easy.

The format of the Swift Park database can be seen in Fig. 3. The database contains four main categories: spotData, spotInfo, userInfo, and users.

The spotData category, seen in 5, contains the fields color, height, width, pass, x, and y. Color determines whether the spot is taken: if the spot is green, it is available; if it is red, it is unavailable. The x and y fields determine the pixel locations on the map of the location of the parking spot. The width and height fields are the size of the parking space. Pass dictates what type of the pass the user must have to park in the spot.

Fig. 3. The Firebase database format.

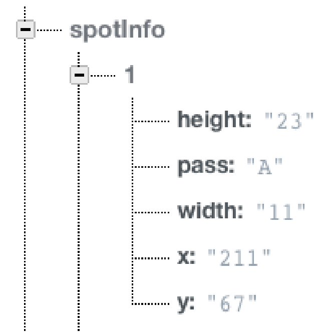

Fig. 4. The format of spotInfo in the database.

The spotInfo category, seen in 4, has all the same categories of spotData except for color. This category is used to craft responses from the controller to the client, while the spotData category is used to by the map on the Swift Park website.

The user category (Fig. 7) contains a credit card, parking pass, password, and unique client ID that is assigned to the user. This controller accesses the user account to see if the person has the correct parking pass to park in the spot it requests. It can also see if the user is able to pay for parking in a paid spot.

The userInfo category, seen in 6, contains a single field that maps the client unique identification to the email address of the corresponding account. This category is used by the controller to access the user's account information via email without having to iterate through every user account looking for the client ID. (This would be necessary as user accounts are located through email, not client ID.)

E. The Controller

The controller is used to mediate between the client and the database. It interacts with the client through the two active IoT protocols to modify the database based

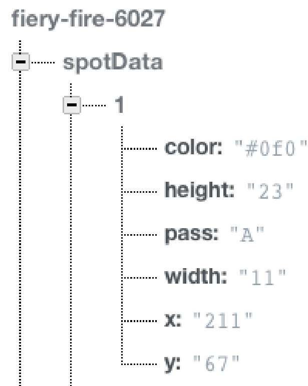

Fig. 5. The format of spotData in the database.

Fig. 6. The format of userInfo in the database.

on the choices of the user, and to inform the client of its options by querying the database to find out about the parking spot the user is occupying.

The controller takes a JSON object from the client through the active client-to-controller IoT protocol. This JSON states what action the client would like to take, the spot the user wants to occupy, and the unique ID of the client. When the controller receives this information it queries the database using an API developed in Python specifically for the controller. The API contains the functions take, release, isAvailable, canPark, and isPayed.

The isAvailable function is executed when the client specifies the request action in the JSON object it provides to the controller. This function gets the spot-Data entry in the database corresponding to the spot specified in the client-sent JSON. The availability of the spot is then determined by the color field of the JSON returned from the database. This information is communicated to the client via the active controller-to-client IoT protocol.

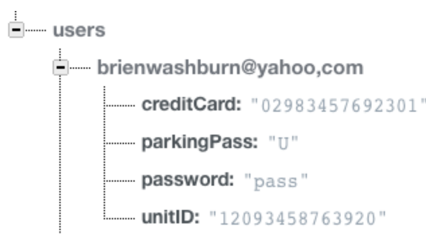

Fig. 7. The format of the users category in the database.

The canPark function also runs when the action specified is request. This function will index the userInfo category in the database. The client ID will be used to find the email address, and that will be used to access the user's account. The account is needed to find the parking of the user. Once the user's parking pass is known, spotInfo is accessed using the spot data specified by the client. The parking pass required by the spot is then retrieved and compared to the parking pass of the user. If the user's privileges set by the parking pass are less than that required by the parking spot, parking privileges are denied. Otherwise, the permission is granted. In both cases the controller communicates the permission status back to the client through active controller-to-client IoT along with the availability data.

The isPayed function executes when the action specified is request. The isPayed function will index the spotInfo category of the database with the spot data provided by the client. The JSON will be parsed to find the pass type. If the pass type is "L", the spot requires payment. This will be communicated back to the user along with the spot availability and the permission to park.

The take function is invoked when the client specifies the claim action. Take indexes the database using the spot data communicated by the client. The controller then modifies the color field of the JSON it received from the database query to red to signal that the spot is no longer available. This JSON is then stored in the database in spotData. The website will be notified of this change and will modify the parking space on the map to show that the spot has been claimed.

The release function executes when the release action is specified by the client to the controller. Release is exactly the same as the take function, except that it changes the color of the spot from red to green, signaling that the spot is now vacant. Again, the website will be notified and will update asynchronously to show that the spot is now empty and available.

F. The Website

The goal of Swift Park is to simplify the entire parking process, and the website contributes to this goal in multiple ways. It allows users to easily access a map of the parking lot to find available parking. It also makes it possible to pay for parking without having to guess the length of time you will be in the lot, carry change for parking meters, or deal with an app that requires a surcharge whenever you pay for additional

time. You can simply make an account registering your unique Swift Park identification, provide the type of parking pass you have, and supply a credit card to be charged when parking in a paid stall.

The two main components of the website are the application and the user accounts. The application mostly uses jQuery to implement and update the map by communicating with the database. The user accounts consist of JavaScript to parse account information entered on the website and update and query the database.

1) General Development: The Swift Park website was developed using JavaScript, the JavaScript library jQuery, CSS, Bootstrap, and HTML5. The website has, at its core, a heavily modified CSS template. The decision was made to use a CSS template for the initial website, but significant changes have been made to suite the project. A search was recently conducted in an attempt to locate the template source so the originators could be properly credited. Unfortunately, the source could not be located, but it is important to note that the website was started from a basic CSS and HTML5 template.

Bootstrap is also heavily employed. Bootstrap is a type of formatting tool used to format a web page using CSS. There is very little CSS in the website that isn't accessed through Bootstrap.

2) Application: The application consists of the map and the software to keep it updated. Updates to the map occur through callbacks setup using the Firebase API, which is built upon jQuery.

Interacting with the Firebase is a fairly simple process that consists of creating a new Firebase using the URL of the Swift Park database. The main categories of the database—namely spotData, spotInfo, userInfo, and users—can be referenced using the child function from the Firebase API with the name of the category as the field.

Once a reference to spotData and spotInfo are created, callbacks can be made by calling the on function on the references. The callbacks of spotData, the category that contains the data representing each parking space, are invoked if a child is added or changed. Both callbacks execute the function drawSpot.

The drawSpot function uses the spotInfo database reference to extract the x and y coordinates of the parking spot, as well as the width, height, and color. All of these attributes are used to modify the map on the Demo-App page of the website. Since the database stores the color corresponding to the parking space's availability, the drawSpot function can be invoked when a spot is taken or released.

3) User Accounts: The user accounts allow the user to provide information about them in relation to the parking lot. A user account holds the user's parking pass, credit-card information, and unique client ID, as well as the user's email and password. All of these values are used to simplify the parking process for the user by informing them when they are allowed to park somewhere and if doing so will cost them money. It will also automate the payment process for them, removing the burden of manually paying repeatedly when time expires or getting a ticket if they don't make it back to the meter in time.

The user-accounts portion of the website was developed primarily using basic JavaScript. Some jQuery was used through the Firebase API.

Account creation takes place on the sign-up page, accessed through the Create Account button in the navbar. This page consists of a Bootstrap form that takes a user's desired email address, password, unit ID (client identification), parking pass, and credit card. When the use clicks the submit button, the form executes the JavaScript function, signup, in scripts.js.

Signup finds the information entered into the form fields based on the division IDs specified in the form. Once the division is acquired, the value field can be accessed, providing each element the user entered. Each element is then passed to a function called setCookie.

The accounts-portion of the website is based on the usage of cookies. Cookies allow information to be stored in the browser until cleared. This means that data doesn't need to be passed from page to page as it is stored in cookies in the browser. It also means that the user can leave the page, come back later, and still have their account information show up in the Demo App because the cookies will be accessed and used to display their information.

The function setCookie takes the name of the cookie provided and the value that it is to be set to. It creates a date object set a year in the future (this means the cookie will last for a year unless cleared manually or written over), concatenates that with the cookie name and value, and sets a desired path that determines where the cookies can be accessed.

Now the user will be redirected to the Demo-App page where their account information will be displayed, along with the application map of the parking lot. The user can navigate through the website using the navbar at the top, and return to the Demo App which displays the map and their account info using the Check

Account button. A user can also sign out of their account using the Sign Out button in the navbar. This will invoke the deleteCookiez function in scripts.js, which, as the name suggests, will clear the cookies corresponding to the user's account.

When a user wants to sign back in after signing out, they are able to do so by entering their email and password in the corresponding boxes in the navbar and clicking Login. This will execute the function cookiez.

Cookiez will find the navbar division which contain the elements entered into the sign-in text fields in the navbar—namely, the email and password fields. Once these divisions are stored and the value of their text boxes are acquired, the function accessAccount is called with the email address and password as parameters.

AccessAccount will use the email address provided to index the user reference from the database. Once the corresponding account has been acquired, the password is verified by comparing it against the password stored in the user account. If the password is validated, the setCookie function is called on each of the items stored in the JSON that was retrieved from the database.

When a web page is selected in the navbar, when the page loads the function loadNavbar is called. This function uses cookies to determine if the user is currently logged in or logged out. If the user is logged in, the navbar is set to provide the Check Account and Sign Out buttons. If the user is logged out, the navbar will provide the email and password text boxes and the Login button, as well as the Create Account page link.

If the user navigates to the Demo-App page on the website, the page will call the displayInformation function. This function calls loadNavbar and populate.

The populate function will locate the division of the account form on the Demo App page and populate them according to the cookies set. If the user is logged out, no information will be display as there aren't any cookies set corresponding to the users account.

The user accounts facilitate the goal of Swift Park, allowing the user to more easily locate and pay for parking without worrying about overpaying for parking, or underpaying and having to hurry and feed the meter to avoid a ticket.

G. Prototype Development

A prototype of the project was developed in the early summer after the switch to QR technology to make certain that the plan for the project was sound. A simple Python client was created using ZBar to analyze a QR code. A Python socket was hard-coded to the controller's URL to transmit the spot data from the QR code. The controller was also hard-coded to access and modify a certain spot in the database (which was very simple at the time, containing a single spot in spotData and spotInfo). The database could then be viewed on the Firebase website to see if it was being modified correctly.

A basic website was created to have a space for the parking-lot map. The map included a text box and two buttons that allowed a user to manually take or release a spot based on the spot information in the text box. This spot information was used to modify the database to reflect the choice made to either take or release a spot. The website listened for changes in the database and updated the map asynchronously based on the changing of the color field in the database.

IV. Testing and Integration

Testing began with the prototype that was developed to validate the project after the switch from RFID to QR technology was made. The simple prototype of the Swift Park system allowed us to verify that each piece was capable of functioning as intended.

1) Client Operation: Each stall is equipped with a QR code that holds information about the stall. Specifically, each QR code contains the URL of the website and a value that identifies the specific parking spot.

The interaction between the car and parking stall was simulated by bringing up a QR code on the laptop and using the camera to take images of the QR code. The camera was slowly pulled back and steadied to allow it to take an image. Each time it was successful the camera was moved back further. The max distance was approximately four feet for a screen-sized QR code, which is a reasonable distance considering the location of the QR code in comparison to the car in the parking space.

The QR codes were generated using qr-code-generator.com. The scanned code was processed in OpenCV and handed off to ZBar. Zbar stored the QR code's information into a file which was printed to the screen and compared against the QR generated.

The active client-to-controller IoT protocol was first tested by creating the JSON object to be sent to the controller and then printing it out on the screen. Once it was shown to be correct, the Python socket was setup using the URL of the server/controller in the CADE lab

on port 2113. The JSON object was then sent using the Python socket and was validated on the controller side.

2) The Controller: The controller, coded in Python, was validated by passing in a JSON object from the command line similar to what the client would send. Once the controller was invoking the correct functions from the custom Python API developed for the controller based on the JSON input, the output meant for the database were tested by printing out the database queries and JSON objects that were to be sent to the database.

3) Website Testing: Testing of the website was primarily a visual task. Changes to the source can be seen easily on the website, so testing consisted of making alterations to the HTML or CSS or JavaScript and then engaging with the new functionality on the website.

The application prototype included a text box at the bottom of the map that took a value corresponding to the parking-stall number contained in the database. Two buttons were next to the text box—one labeled ”Take” and the other ”Release.”

The ”Take” button would use the stall number entered into the text box to modify the database so it showed a car parked in the stall. The database could then be pulled up to check if the change was made, and it could be confirmed by looking at the map to see if the spot corresponding to the number entered had changed from red to green.

The ”Release” button would again use the number entered to change the corresponding parking space's entry to mark the car as having left the spot. This could then be confirmed by checking map to see the if the spot had changed from red to green.

The user accounts were validated in a similar fashion as the application. Account creation was easy to verify as the account data entered by the user is shown on the Demo-App page after they create their account. The same thing happens when a user logs in. The database can be verified easily visually.

The development and testing of the website was made easier by the developer tools embedded in Safari. Safari isn't the best browser, but the debugging tools it includes that allow you to see the source code of a web page, set break points, and step through code was a great boon during the web-development portion of the project.

A. Integration

Integration in Swift Park was a fairly simple task. This is because each major piece of the project is fairly discrete, meaning that once the client or controller or website worked appropriately, there weren't many interactions that could negatively impact the functionality of those connected major modules. The data passed from client to controller and vice versa is simply a JSON with no more than three fields. The controller's interaction with the database is equally simple once the Python code to index the database was worked out. The controller provides the database with a JSON object that it places at the path provided. Lastly, the interaction between the website and the database is similar to the controller-database interaction. The database is being queried using JavaScript to retrieve JSON objects, and JavaScript is used to push JSONs to the database.

The original integration occurred with the prototype. After the switch was made from RFID, the prototype began with scanning and processing QR codes in the client. Once the QR codes were shown to be correct, the client began compiling the JSON outputs to the controller and printing them out.

After the JSON outputs of the client were tested and shown to be correct, the client was connected to the controller using a Python socket. To validate the connection between the controller and client, the JSON received by the client was printed out and checked against the JSON sent by the client. The controller-to-client protocol was tested in much the same way.

The controller was then tested by seeing if the custom API's functions were being correctly executed based on the JSON input from the client. Each method printed out a sentence stating that it had been invoked. Once all the methods were being called correctly, the controller was connected to the database. Validating changes in the database made by the controller was simple due to the easy interface of the Firebase website.

Lastly, the website was connected to the database. Again, validating the interactions between the database and the website was simple because there are many visual indicators that make it clear whether the modules are connected correctly.

The significant modularity of Swift Park made it easy to identify and resolve issues when connecting any of the larger pieces together of the project together. Validation within a major module was generally more difficult to do.

V. Acknowledgments

There are a couple of resources that should be acknowledged for providing significant support to the project.

1) SkyRFID: SkyRFID saved us a lot of time and energy by informing us that it would not be possible to implement Swift Park as planned with RFID technology. They motivated the switch to QR technology, which was a great fit for Swift Park.

2) W3Schools: W3Schools, a website dedicated to programming tutorials, provided significant support in learning web development. None of the team members had any exposure to web development before this project, so W3Schools was an invaluable resource in learning JavaScript, HTML5, jQuery, CSS, Bootstrap, and about web development in general.

VI. Conclusion

The world is becoming increasingly connected, and the internet of things is the next logical step in the evolution of the internet. Searching for convenient parking is an unnecessary activity no one wants to engage in, and an intelligent lot-monitoring system can utilize the internet of things to decrease the time and effort required to find and pay for parking. Other lot-monitoring systems oftentimes require the users to drive into the lot before it is possible to determine if there is any available parking. Swift Park removes this inconvenience through a free website that allows users to see if the lot has available parking long before they are on location, and provides users a way to pay for parking without the hassle of using parking meters or apps where you have to guess how long you will be parked.

Swift Park was an interesting and enriching senior project for us because it required an understanding of modern networking infrastructure, it necessitated the design and implementation of custom software, it required us to learn web development, and we had to be able to make use existing hardware and software to make a new product. The project was software-heavy, but we were able to address this issue in part by creating a headless display for the Raspberry Pi consisting of LEDs and switches that allowed the client to interact in a meaningful way with the Swift Park system.

The project also required us to evaluate and mitigate risks as they arose. We were originally going to use RFID technology instead of QR codes, but we took the initiative and spoke with people in the industry that advised us that it would be very difficult to make the RFID technology work with the project. With this information we made the decision to move to QR codes, which proved to be a great fit for the project.

Swift Park as an application for the internet of things demonstrates just how powerful the internet can be. With QR technology, there are so many possible applications that can simplify and enrich our lives as data capture through cameras becomes more and more common. RFID will also be a great avenue of exploration as the technology matures and allows for longer range scanning without significant interference. The internet has been a wonderful tool to share information, and the next stage in its evolution, the internet of things, will enhance the our environment as every item will be capable of communicating and sharing with the world around it.

References

[1] G. Kortuem, F. Kawsar, D. Fitton, and V. Sundramoorthy, "Smart objects as building blocks for the internet of things," *Internet Computing, IEEE*, vol. 14, no. 1, pp. 44–51, Jan. 2010. [Online]. Available: http://dx.doi.org/10.1109/mic.2009.143

[2] "Individual parking space monitoring - sensors." [Online]. Available: http://www.swarco.com/sts-en/Products-Services/Parking/Detection/Individual-Parking-Space-Monitoring-Sensors

[3] K. Grifantini, "Find a parking space online," Jul. 2008. [Online]. Available: http://www.technologyreview.com/news/410505/find-a-parking-space-online/