Parking Status

A Full Stack Web System for Managing Parking Lot Data

Dennali Grissom

COS 492 - Covenant College

December 6, 2021

https://github.com/dennalig/ParkingStatus

Table of Contents

Introduction

Parking Status is a web application for managing an institution's given parking lot information, keeping track of certain statuses or reservations that a parking lot may have throughout a given day or week, and presenting that information clearly and purposefully to a guest or someone who may not be familiar with the parking procedures of said institution. Parking Status is meant to allow clear communication between the institution's parking procedures and a given person to whom those parking procedures may concern (such as a visiting guest or a new student, for example). Since Parking Status is a web application, any institution that would want to use it would be able to host it onto their server and anyone would be able to access that institution's version of the solution via a Uniform Resource Locator (URL). My personal and academic goals for creating the Parking Status system were to a) implement a practical solution to a real-world problem using programming fundamentals that I had learned throughout my time as an undergraduate computer science student, b) utilize what I had learned from internship experience (in regards to the software development lifecycle, development methodologies, and programming frameworks) orchestrate a well-thought out design and development process that would allow for the system to be maintainable, c) get out of my 'comfort zone' by practicing web development to create an intuitive frontend, d) utilizing my programming experience to build a full stack solution that harmonizes a backend, frontend, and database, and e) create a solution that could be tailored to any institution for its purposes.

The Issue

Issue Statement

According to a 2019 study, an American driver spends about 17 hours and $345 in wasted fuel and emissions per year just in the act of searching for a parking spot. From the 16 major U.S. cities from this same study, there was a collective 1.4 billion dollars in annual parking ticket revenue.[1] There have been many instances where knowing when and where to park has always been a major stressor to a seemingly menial situation. Whether it be a family attempting to visit a highly populated event, an employee at a company facility knowing which areas are designated for him or her, or a student wondering if there are closer parking options for the weekend, it can be outright difficult and an unclear venture to constantly be concerned about. With the validity of street signs varying from place to place, a guest not being 'in the loop' on a certain email that informs people to not park at a place due to an event, and the simple ambiguity of a guest visiting a new location, the lack of clarity can cause confusion. There have been certain attempts at helping clarify when parking was allowed that were useful, only if one knows where to look. Covenant College for example, utilizes their Safety and Security department by having them both send out informational emails to respective students as well as provide descriptive maps of the campus in regards to this. However, this method of presenting the data does not necessarily help clear up the ambiguity as much as it could. For example, I have had many peers that have gotten parking tickets due to an unknown event happening that affected the location of their current vehicle. Overcrowding of certain parking areas has taken place due to the ambiguity on what places are and are not allowed. Following along with the proposed solution of Covenant College's Safety and Security, the means of emails requires that anyone that is parking on campus be connected to that email list, which is not realistic. It also requires those that *are*

[1] Lisa M. "Research Proposal: Parking Tickets" *Wonder*

connected to the email list to constantly sift through their cluttered inboxes to check if Safety and Security did, in fact, send the email. The means of campus maps, while inclusive to the non-attached Covenant visitor, only has the capability of statically storing that information with no easy way to update it without the illustration of another map. The map solution also does not take into account special and temporary events that may take place within an institution such as a college's sporting event, homecoming, or a visitor's weekend and once again, the institution now needs to send out those exclusive emails in an attempt to communicate the update. The information about these matters should not be as difficult or seemingly exclusive as they are in order for a common person to be able to know.

Issue Solution

Parking Status strives to resolve this issue of ambiguity and miscommunication by allowing institutions to generate a central, organized, and reliable source on the statuses of its parking lots. It is an electronic source that displays all of the parking lots that an institution may have, information about the parking lot, and the current status of that parking lot. This will allow individuals to be able to visit the site that the application is hosted on, see given parking lot options and see their current statuses that update based on the current time and day of the institution's designated timezone. Whether the guest is fully integrated within the institution (like that of a student or an employee) or just a visitor, this allows for anyone to know when and where parking is available just by accessing a given link. An administrator user (someone in control of the institution's Parking Status system) will be able to enter in the parking lots, parking statuses, and special events that the institution wants to keep track of, as well as certain corresponding information sets for a given lot, status, or special event that the institution desires

to store and keep record of. A non-administrator end user (a visitor or person viewing the Parking Status system, but with no permissions to alter the data or information) will be able to visit this link and see a list of given parking lots, a description of the given parking lot, and a certain assigned status for the parking lot that gets updated once the time changes. For the casual institutional guest, Parking Status provides for a quick lookup of this information by displaying it on one singular page with the hopes that it would negate the confusing aspects of not knowing where to park. For the institution that is implementing Parking Status, it provides a manner of quickly creating, editing, and deleting parking lot data for the institution itself as well as any specific occasions that the institution wants to keep track of.

The Covenant College Campus as an Example

Parking Status is designed and intended to be used for any institution that has specific parking reservations that it may enforce throughout a given day and/or week. Some very basic examples or use cases of what Parking Status was designed for were the likes of a large facility, a parking deck, designated sections of a crowded downtown area, and a college campus. The initial idea for proposing Parking Status as a solution came from my reflection on how confusing and disorganized it can seem, at times, to understand parking at a campus like that of Lookout Mountain, Georgia's Covenant College. Throughout this description of the Parking Status solution, I will be constantly using aspects of Covenant College's campus as a simple, yet specific, use case for how certain aspects of Parking Status ought to operate or have been implemented in the source code.

High-Level Implementation

Administrative Features

We will start by discussing the features that a system administrator would be allowed to utilize in order to migrate and maintain their parking lot information and dynamics to the Parking Status system.

Firstly, in order to add an administrative user (admin) to the system, one needs to register as an administrator. The user is presented with a navigation bar (navbar) and the homepage. On the left there is a link with the title of 'Parking Status' that redirects the user to the home page. On the right side are two buttons: 'Sign Up' for registering a new user and 'Admin Login' for logging in as an administrator. To allow for admin use, Parking Status provides a simple signup system where a user can provide their email address and their desired password. A user that desires to be an administrator *must* enter these details in order to be stored as an administrator on the solution. Once the administrator is signed up, he or she must officially login by clicking on the 'Admin Login' button on the navbar. Here the user enters their email and password credentials, and if successful, the administrator is registered as being 'logged in', allowing the Parking Status system to open up the administrator features to the user.

Admin Login Feature

Once the administrator is logged in, he or she will see that the navbar has slightly changed. ‘Admin Login’ will now be changed to ‘Logout’ and under that are three buttons labeled as ‘Lots’, ‘Status’, and ‘Status Event’. Also, under the ‘Parking Status’ link on the left, the user will see a button labeled as ‘Edit TimeZone’. ‘Lots’, ‘Status’, ‘Status Event’, and ‘Edit TimeZone’ house all of the administrative features that are available. The main three objects, or models, that an administrator will be interacting with will be that of a ‘Lot’ (meant to represent a singular parking lot, so for Covenant College, Sanderson Lot would be represented here), ‘Status’ (meant to represent a singular status that can be assigned to parking lot at a given time, the status of ‘Faculty Parking Only’ for Covenant College would be represented here), and a ‘Status Event’ (meant to represent a special or temporary event that does not happen regularly

and can be assigned a specific status and parking lot for a given date and time, the event of ‘Homecoming’ at Covenant College would be represented here).

The ‘Lot’, ‘Status’, and ‘Status Event’ buttons are the links (an ‘object selector’) to which an admin user can create a new instance of one of these three main objects, see a list of all of one of the three main objects at a time (a list of all the stored Lots, Statuses, or StatusEvents), and an option to select an existing instance of an object in order to edit or delete it. On the top left corner of each object ‘object selector’ is a button labeled ‘Create (lot, status, or statusevent)’. Clicking this button allows the user to create a new instance of that given object.

Lot Selector Feature

➢ Lot

- A ‘Lot’ object stores a representation of a parking lot that an institution may have.
- For a ‘Lot’ object, the attributes that an admin can dictate are: a 'Lot' id, a 'Lot' name, a description, and a ‘lot status schedule’. The 'Lot' id is a unique numerical identifier that is a number greater than or equal to one and is used to uniquely

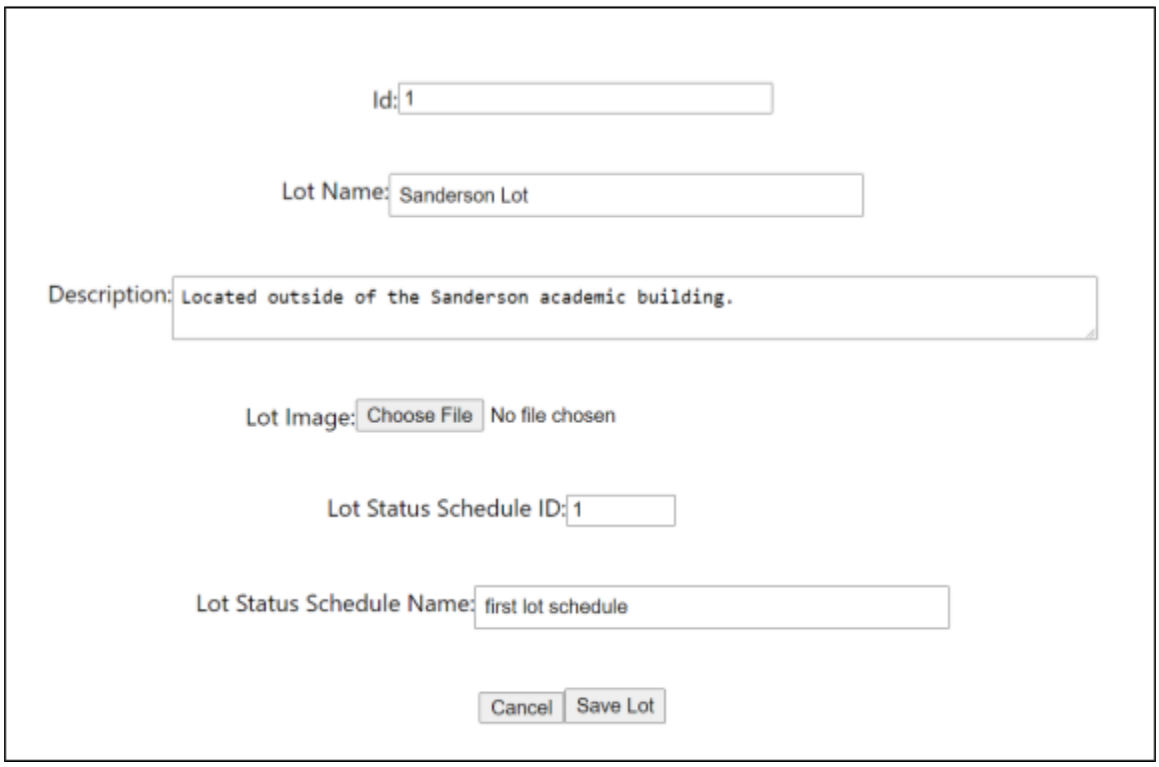
differentiate between parking lot objects. Two 'Lot' objects are not allowed to have the same 'Lot' id. The 'Lot' name is simply the given name of the parking lot that the admin wishes to give it in the system, such as ‘Sanderson Lot’. This attribute does not need to be unique. The description is simply an area where an admin can provide a description of the parking lot such as an explanation, location description, or even a maximum capacity. For the Covenant College Sanderson Lot example, a description could be ‘This parking lot is located next to the Sanderson academic building and near the Maclellan & Rymer dormitory building’.

- The ‘lot status schedule’ is a feature that was designed with the intention of allowing admins to be able to designate a weekly schedule for the parking lot to follow. Each ‘lot status schedule’ is unique to each 'Lot' object and cannot be reassigned. This feature allows the admin user to specify certain attributes that would make up a ‘lot status schedule’ for a given lot: the ‘lot status schedule’ id which is another unique numerical identifier that must at least have the value of ‘1’, a ‘lot status schedule’ name that provides a name for the schedule (eg. ‘Sanderson Lot Schedule’), and multiple instances of what the system names as a ‘lot status schedule date’.
- A ‘lot status schedule date’ is a specific instance where an admin user can specify a block of time that a parking 'Lot' object can be assigned a certain 'Status' object (see the ‘Status’ subsection for more details on this object). The data that is stored within a ‘lot status schedule date’ is meant to be flexible to the needs of parking lot statuses. It is based on a generic seven-day week (Sunday through Saturday)

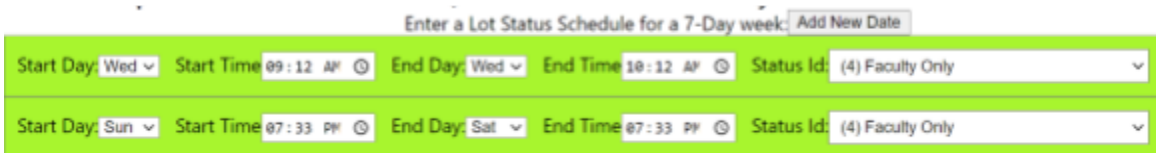
and a 24-hour period for each day. Each ‘lot status schedule date’ allows for the admin to specify a ‘Start Day’ which is any option between ‘Sun’ and ‘Sat’, a corresponding ‘Start Time’ which is any option between 12:00 A.M. to 11:59 P.M., a matching ‘End Day’ and ‘End Time’ with the same dynamics as ‘Start Day’ and ‘Start Time’ but intended for dates and times that are later than the designated ‘Start Day’ and ‘Start Time’, and a list of options to designate which ‘Status’ object that the admin wants to assign to that date using the ‘Status’ object’s unique id.

- An admin can assign as many ‘lot status schedule dates’ to a ‘lot status schedule’ as desired. Also, each ‘lot status schedule date’ can designate a block of time that lasts either within one day for a certain time or across multiple days within a given week for a certain time.
- Ultimately, the ‘lot status schedule dates’ that the admin specifies are blocks of information that will be communicated to the non-admin user when he or she wishes to see which parking lots may or may not be available
- Following with the Covenant College Sanderson Lot example, an admin could specify that the ‘Sanderson Lot Schedule’ has a ‘lot status schedule date’ that starts on Monday at 6:00 A.M., ends on Monday at 5:00 P.M. , and has a ‘Status’ id that references a parking status for indicating the status of ‘only faculty can park here

Lot Create/Edit Feature



Lot Status Schedule Date Feature



➢ Status

- A ‘Status’ object represents a certain status or state that an institution may want to assign to a certain parking lot at a particular time.
- For a ‘Status’ object, the attributes that an admin can dictate are: a ‘Status’ id, a ‘Status’ name, a ‘Status’ color, and a description. A ‘Status’ id is a unique numerical identifier that is a number greater than or equal to one and is used to

uniquely differentiate between parking ‘Status’ objects. Two ‘Status’ objects cannot have the same id. A ‘Status’ name is simply the name of the ‘Status’ object that the admin wishes to designate such as ‘Faculty Only’. While seemingly insignificant, the ‘Status’ color is a very important feature within the system. The admin can specify a specific color for the given ‘Status’ that a non-admin user will be able to quickly identify as being associated with that status. All color options are formatted to hexadecimal. For the ‘Faculty Only’ example, a color of ‘ff0000’ (a shade of red) can be specified. A description simply allows the admin to provide an explanation of the given ‘Status’ such as ‘Only faculty can park here. Students and guests are not permitted to park here.’.

- Status Event
    - A ‘Status Event’ object represents a temporary occasion or event that an institution may have where routine parking lot schedules are bypassed in order to accommodate the new specifications of the occasion.
    - For the ‘Status Event’ object, an admin can dictate: a ‘Status Event’ id, a description, an assigned ‘Status’ id, and multiple instances of a ‘Status Event date’. A ‘Status Event’ id is a unique numerical identifier that is greater than or equal to ‘1’ for a ‘Status Event’ object. A description can act as either the name or an explanation of the particular ‘Status Event’ object. For the Covenant College example, ‘Homecoming’ could serve as a ‘Status Event’ description. The selected ‘Status’ id references a particular ‘Status’ object.
    - A ‘Status Event date’ instance operates in the same vein as a ‘lot status schedule date’ (see the ‘Lot’ subsection for an explanation). However, instead of ‘Start’

and ‘End’ days, we have specific calendar dates (‘Start’ and ‘End’ dates) in order to specify that a ‘Status Event’ is temporary and not a regular part of the weekly ‘lot status schedule’ rotation. Also, instead of selecting a designated ‘Status’ id in the ‘Status Event date’ instance, an admin user must specify a ‘Lot’ id that the institution wants the ‘Status Event’ to be assigned to instead.

- A ‘Status Event’ object can have many ‘Status Event dates’, each with a differing assigned ‘Start’ and ‘End’ time, ‘Start’ and ‘End’ date, and parking lot, .
- Following with the Covenant College Sanderson Lot and ‘Homecoming’ example, the ‘Status Event’ of ‘Homecoming’ could be assigned a status of ‘Alumni Only’ and then contain a ‘Status Event date’ that spans from September 17th, 2021 at 6:00 A.M. to September 18th, 2021 at 5:00 P.M. and be assigned to the ‘Sanderson Lot’ parking lot object. Instead of the regularly scheduled ‘Faculty Only’ ‘lot status schedule date’ that is assigned to ‘Sanderson Lot’, it will display its current status as ‘Alumni Only’ for any overlapping time between the ‘Homecoming’ ‘Status Event date’ and the ‘Sanderson Lot’ ‘lot status schedule date’.
- In the event that a ‘Status Event date’ and a ‘lot status schedule date’ conflict with the same designated times and parking lots, the ‘Status Event date’ will take priority if the calendar date specified overlaps.

➢ Edit Timezone

- This feature allows admins to specify which timezone Parking Status is operating on when dealing with ‘lot status schedules’ and ‘status events’, such as Eastern Standard Time (EST) or Central Standard Time (CST). Specifying a different

timezone will cause Parking Status to access the current time of that specified timezone and display the ‘lot status schedules’ and ‘status events’ accordingly.

- The main intention behind this feature is to not rely on a local date and time that is stored on a user’s (admin or non-admin) machine as that would incorrectly alter how the schedule information would be presented to anyone outside of that timezone.

Along with being able to create an instance of each of these three objects, an admin user can also edit and delete them.

In order to edit a certain object instance, the admin must select the desired object instance from its appropriate object selector, (e.g. select the ‘Homecoming’ ‘Status Event’ from the ‘Status Event’ object selector). Once selected, the admin clicks ‘edit’ and the admin user is brought to a page that mimics the ‘Create object’ page with the selected attributes of that object automatically filled in. An admin user cannot edit unique numerical identifiers, however, all other attributes are mutable (e.g. revise the ‘Homecoming’ ‘Status Event’ to have a color of ‘#0000FF’ (a shade of blue) instead of red.). Upon editing a ‘lot status schedule’ on a ‘Lot’ object or ‘Status Event dates’ on a ‘Status Event’ object, the preexisting dates are also automatically autofilled, however the admin user must manually click ‘save’ on those particular dates again.

In order to delete a certain object instance, the admin must select the desired object instance from its appropriate object selector, and then click ‘delete’ (e.g. in order to delete the ‘Sanderson Lot’ instance, an admin would need to select the ‘Lot’ list, select the ‘Sanderson Lot’ item within that list, click ‘delete’, and then verify their decision to delete it by clicking ‘Yes’ when asked ‘Are you sure you want to delete the Lot?’. This will result in ‘Sanderson Lot’ being

removed from the list of ‘Lot’ objects.). Upon deleting a ‘Lot’ instance, all corresponding ‘Status Event dates’ within the ‘Status Event’ objects that were assigned to that parking lot will also be removed. Upon deleting a ‘Status’ instance, all corresponding ‘lot status schedule dates’ within ‘Lot’ objects that were assigned to that status will be removed as well as all corresponding ‘Status Event’ objects that were assigned that particular status.

Non-Administrative Features

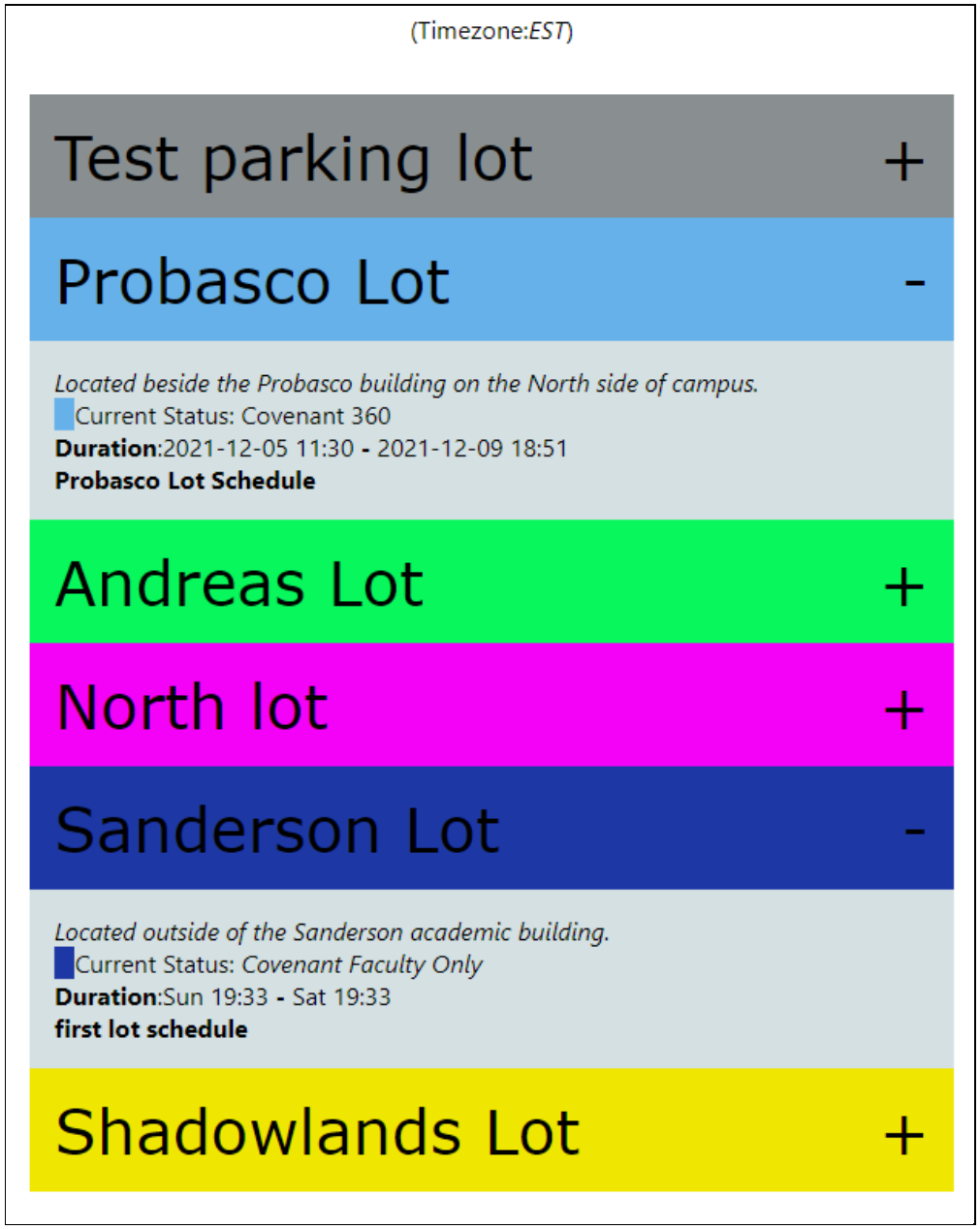
While Parking Status does not have any *explicit* non-admin features, I think it was important for the solution to be able to differentiate between what an admin has access to within the solution compared to a non-admin user. For the scope of the Parking Status application, I am defining a non-admin user as anyone who would be accessing the Parking Status application in order to view the status of a parking lot to determine if he or she is allowed to park there, but he or she does not have access rights to be able to edit any of the data (such as a parking lot, schedule, status, status event, or designated timezone) that is presented to him or her. This would be the likes of a campus visitor, a spectator at a sporting event, or a downtown commuter, for example. Essentially, the only Parking Status feature that a non-admin user has access to is the homepage. The option to view, edit, or delete ‘Lot’ instances, ‘Status’ instances, ‘Status Event’ instances, or the designated timezone are inaccessible. In order for the Parking Status to render those corresponding links and buttons within the navbar, the user must be logged in as an admin user. I consider the denial of access to the designated admin features as a non-admin feature.

- Home Page
    - Parking Status’s home page is where the main appeal of the solution is presented. This page mends together all of the stored data across the ‘Lot’, ‘Status’, and

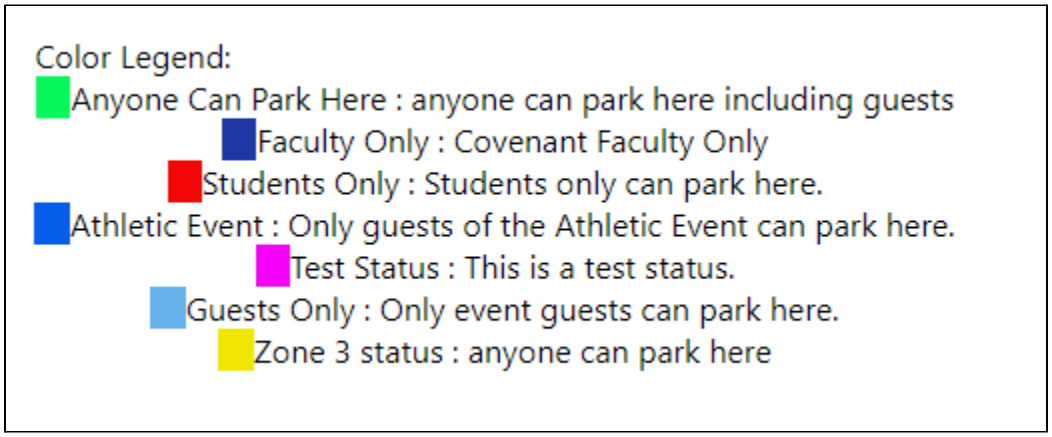
‘Status Event’ objects as well as the designated time zone. Firstly, there is text to indicate what current time zone the admin has specified the system to be operating in, (eg. ‘Time Zone: EST’). Secondly, This page contains an accordion-style User-Interface (UI) component that lists each existing parking lot within the system. Each lot entry in the component will be color-coded based on the current ‘Status’ that is assigned to the parking lot (whether it be from the regularly scheduled ‘lot status schedule’ or the specialty 'Status Event'). As the assigned statuses change for a given lot, the colors of the accordion entries will change upon refreshing the page. If a certain parking lot currently does not have an assigned 'Status' to it, the default color of that entry will be a shade of gray and its assigned ‘Status’ will be ‘none’. Thirdly, upon clicking on a lot entry in the accordion, that specified entry will expand with more information about that particular lot such as the parking lot description, the current status name (If this status comes from a ‘lot status schedule’, it will just display the status name. If this status comes from a 'Status Event', it will just display the 'Status Event' description), the duration of that current status, and the name of the assigned parking lot schedule. Clicking on these lot entries in the accordion will toggle them as being ‘expanded’ or ‘unexpanded’ accordingly. Lastly, the home page comes with a designated ‘color legend’. Clicking the button labeled ‘color legend’ on the left will display a section above the accordion with all of the possible status colors, their corresponding status names, and corresponding descriptions of that status. This allows for any ambiguity on colors to the non-admin user to be clarified, especially when trying to differentiate between a regularly scheduled

status and a temporary ‘Status Event’ status. The non-admin user can toggle this as visible or non visible by clicking either the ‘Color Legend’ button or the ‘X’ button that is presented below that.

Home Page Lot Accordion



Home Page Color Legend



Low-Level Implementation

In terms of low-level parts, Parking Status has many moving practical and abstract pieces that must come together in order for the solution to be implemented in a manner that is intuitive and efficient. I would consider the important aspects of Parking Status's low-level implementations to be that of the domain models, database, backend, and frontend. Each of these aspects have a tree of other compartmentalized aspects that were involved in the implementation.

Domain Models

In software development, domain modeling is "...a way to describe and model real world entities and the relationships between them…"[2]. Making an adequate domain model is something that is crucial to the implementation of a real-world solution. During the early stages of developing Parking Status, there were many aspects that had the potential to easily be mismodeled for the rest of the application. Essentially, domain modeling allows one to conceptually sketch out the structure of the entities and their relationships that will ultimately affect how one shapes the rest of their application, including the database, backend, and frontend.

[2] "Domain Modeling" *Scaled Agile*, https://www.scaledagileframework.com/domain-modeling/

A poor domain model can hinder the progress made on a project, especially during the latter stages of development, when there are realizations that the current shape of the data is not suitable for a particular section of the solution that one is trying to solve. During the Summer of 2021, this is exactly what happened on my internship team at CGI Inc., where we realized that the scope of our attempts to modularize instances of workflows would not be manageable when a later team would need to implement those instances for different purposes. It is important to be able to conceptualize how this data should be outlined and stored as well as whether or not those current design decisions are sustainable. This is not to say that one needs *perfect* domain models, because that is also an impossible feat. In the same manner that code can always be written more efficiently, domain modeling could be designed in a more ideal way. However, as humans we realize our finitude in many areas, and domain modeling is no exception.

During the latter stages of Parking Status, there were a few times where my current domain model for a particular object was not as suitable as it needed to be. In those times, I needed to completely revise both the backend and databases respectively. Examples of this can be seen in my decision to include the ‘Status Event Id’ field within the ‘Status Event Date’ model or when I realized that a ‘Status’ object should only have the capacity to store colors as a hexadecimal string and not allow for generic named colors such as ‘green’ or ‘pink’.

So in terms of Parking Status, what did domain modeling look like? I needed to be able to sketch out a sustainable manner in which I needed to keep track of, store, update, and delete data that would need to be presented to the end user. To start off, I knew I would need some sort of ‘Lot’ object to represent a parking lot and I knew I needed an interchangeable ‘Status’ object that could be used across each of the ‘Lot’ objects. So those are two abstract models right there, but what about scheduling, specific dates, times, and events? What exact fields do the ‘Lot’ and

‘Status’ models need? It is this list of abstract concerns that are ‘not so obvious’ in terms of data that I needed to represent well within the Parking Status solution so that data usage would be ideal and efficient. Ultimately, the domain model structure that I settled on is listed below (If the domain model structure is too verbose to digest, I have included a simplification of this structure below the outline).

- ➢ Structure
  - (**Lot**) The ‘**Lot**’ model contains the following fields
    - *LotID*: **integer** (required) - A unique, numerical identifier
    - *LotName*: **string** (required) - A non unique display name
    - *LotDescription*: **string** (not required)- A non unique display for the explanation of a lot
    - *LotStatusSchedule:* object (not required)
  - Each ‘**Lot**’ model has the option to contain a single ‘**LotStatusSchedule**’ model. This model keeps track of the regular rotating schedule that a ‘**Lot**’ object may have for a given week
  - (**Lot**) The ‘**LotStatusSchedule**’ model contains the following fields
    - *LotStatusScheduleId*: **integer** (required) - A unique, numerical identifier of the schedule
    - *Name*: **string** (not required) - A non unique name for the schedule
    - *LotId*: **integer** (required) - A unique reference to the lot that the schedule was created for
    - *LotStatusScheduleDates*: **A list of objects** (not required)

- Each '**LotStatusSchedule**' model has the option to contain multiple 'LotStatusScheduleDate' models. The '**LotStatusScheduleDate**' model is used to represent specific blocks of time throughout a generic week in which a certain '**Status**' will be assigned to that particular parking lot based on the current time of the designated time zone.
- (**Lot**) The '**LotStatusScheduleDate**' model contains the following fields
  - *lotStatusScheduleDateId*: **integer** (required) - A unique, numerical identifier of that particular date object
  - *lotStatusScheduleId*: **integer** (required) - A reference to the LotStatusSchedule that the date object is related to
  - *statusId*: **integer** (required) - A reference to the 'Status' object that is assigned to the particular date object
  - *startTime*: **string** (required) - A string that specifies the starting day (Sun. through Sat.) and starting time (00:00 through 23:59) that the 'LotStatusScheduleDate' object will be marked as the active status on the 'Lot' object.
  - *endTime*: **string** (required) - A string that specifies the ending day and ending time that the 'LotStatusScheduleDate' object will be unmarked as being the active status of the 'Lot' object
- (**Status**) The '**Status**' model contains the following fields
  - *statusId*: **integer** (required)- A unique, numerical identifier of that particular Status object
  - *name*: **string** (required)- A non unique display name

        - *color*: **string** (required) - A non unique string for storing the hexadecimal color value of the status for homepage display purposes.
        - *description*: **string** (not required)- A non unique display for the explanation of a status
    - (**StatusEvent**) The '**StatusEvent**' model contains the following fields
        - *StatusEventId*: **integer** (required) - A unique, numerical identifier of that particular StatusEvent object
        - *StatusId*: **integer** (required) - A reference to the 'Status' object that is assigned to the particular StatusEventDate object
        - *Description*: **string** (not required) - A non unique display for the explanation of a StatusEvent and/or to double as a 'StatusEvent' name
        - *StatusEventDates*: **A list of objects** (not required)
        - Each '**StatusEvent**' model has the option to contain multiple 'StatusEventDate' models. The '**StatusEventDate**' model is used to represent specific blocks of time throughout a given calendar in which a certain '**Lot**' will be assigned to that particular parking '**StatusEvent**' based on the current time of the designated time zone. These are similar to '**LotStatusScheduleDates**' within a '**LotStatusSchedule**' however, instead of specifying start and end 'days', it specifies start and end 'dates' with specific calendar dates being stored. Also instead of referencing a specific '**Status**' object, it references a particular '**Lot**' object that is affected by the StatusEvent.
    - (**StatusEvent**) Each '**StatusEventDate**' model contains the following fields

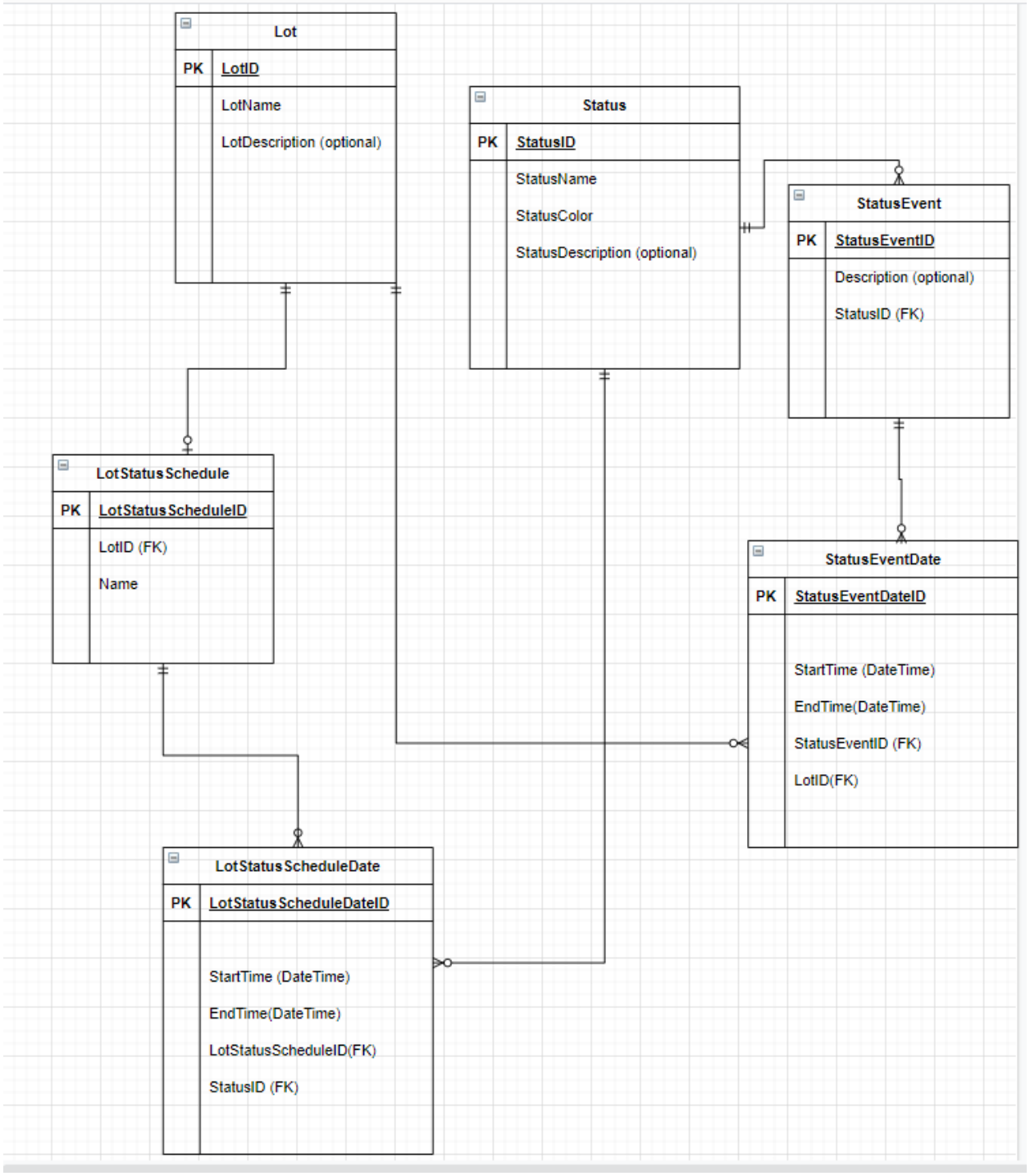
- *statusEventDateId*: **integer** (required)- A unique, numerical identifier of that particular StatusEventDate object
- *statusEventId*: **integer** (required) - A reference to the 'Status' object that is assigned to that particular StatusEvent object
- *lotId*: **integer** (required) - A reference to the 'Lot' object that is assigned to that particular StatusEvent object for a particular block of time within a calendar
- *startTime*: **string** (required) - A string that specifies the starting date (YYYY-MM-DD) and starting time (00:00 through 23:59) that the 'Lot' object will be marked as the active status on the 'StatusEvent' object.
- *endTime*: **string** (required) - A string that specifies the ending date (YYYY-MM-DD) and ending time (00:00 through 23:59) that the 'Lot' object will be unmarked as the active status on the 'StatusEvent' object.

➢ Structure Simplification

To summarize and simplify, each Lot object can have only 1 LotStatusSchedule object and each LotStatusSchedule object can have 0 or many LotStatusScheduleDate objects. Each LotStatusScheduleDate object references a particular Status object. Each Status object stands alone and does not have any embedded objects in it nor does it have any references to any other objects. Each StatusEvent object references a 'Status' object and can have 0 or many StatusEventDate objects. Each StatusEventDate object references a particular 'Lot' object.

Database

The database is integral to the actual information that Parking Status utilizes in its Create, Read, Update, and Delete (CRUD) capabilities as well as its authentication capabilities for a given admin user. Given the outline of the domain models, transferring that over to a database was straightforward. The Schema for Parking Status has been set up as follows.



One table that is not listed here but added near the later stages of production was an 'adminusers' table storing the emails addresses and passwords (hashed, of course) of admin users.

The particular technologies that I used for the database were: PostgreSQL server and app.diagrams.net (for sketching out the database schema). The reason for selecting PostgreSQL was due to its simple and easy integration into my selected backend of Spring Boot. With the implementation of certain PostgreSQL and standard Spring Boot Java Database Connectivity (JDBC) dependencies, making conditional queries on the backend in specified Data Access Service classes became simple. For auto-incrementing values such as numerical ids, PostgreSQL requires the usage of its 'Serial' data type to implement this. I used the 'Serial' data type on both the LotStatusScheduleDate and StatusEventDate tables, as an update on those tables requires a deletion of existing entries to make way for new ones.

Backend

Parking Status's backend was developed using Java's Spring Boot framework. Coined as the "...world's most popular Java framework."[3], Spring was initially launched by Rod Johnson in 2002. Several different types of frameworks, such as Spring Boot, have been developed on top of Spring. Spring Boot allows for many ideal features such as an embedded server and automatic configuration to Spring functionality[4]. For these reasons, Spring Boot is an extremely popular choice for enterprise applications.

There are a few important pieces to any Spring Boot application: an initializer, a dependency file (usually named 'pom.xml'), an 'application.properties' file, and a starter Java file with a reference to the SpringApplication class. The initializer is a website where one would start their Spring Boot project titled 'Spring Initializr' and specify specifics such as a Maven or

[3] "Why Spring?" *VMware, Inc.*, https://spring.io/why-spring
[4] "A Comparison Between Spring and Spring Boot" *Baeldung*, https://www.baeldung.com/spring-vs-spring-boot

Gradle-type project, the Java version, and pre-installed dependencies[5]. For Parking Status, I selected the Maven option, Java 11, and named the project as ParkingStatusApplication. Once downloaded and extracted to a desired folder, one can import the project to their favorite IDE for Java (Intellij, for me, as it provides quick recognition of new dependencies in the Pom file as well as ideal building and running tools). The Pom file stores all of the desired dependencies that may be useful for certain uses of the project. For Parking Status, I used several, but some of the more vital ones were Postgresql for database connectivity, Spring-web for web configurations, and Guava from Google for implementing the SHA-256 hashing function that was used to store admin user passwords. In the 'application.properties' file, it starts out empty however this is where I had to specify database locations for backend connectivity. I had to provide three values of spring.datasource (url, username, and password) in order for JDBC to be able to find the PostgreSQL database.

Another defining Spring Boot application feature is the usage of annotations. These are denoted with an At sign ('@') and usually can be found above Java classes, a global field variable, and methods with varying descriptors on them such as '@SpringBootApplicaiton', '@RestController', '@Autowired', or '@GetMapping'. Each descriptor provides a different type of usage for the Java attribute that it is annotated over. These are pivotal in making a Spring Boot backend that is responsive, capable of communicating with a given frontend, and handling requests (GET, PUT, POST, and DELETE).

In a oversimplified manner, the backend of Parking Status is a set of four parent endpoints ('lots', 'status', 'statusevents', and 'adminuser') that can be consumed via a request on the frontend : an Application Programming Interface (API).

[5] "spring initializr" *VMware, Inc.*, https://start.spring.io/

The application is architected in a Model-View-Controller-like style (MVC)[6] where there are varying layers in the backend that perform certain responsibilities. To demonstrate how this architecture is implemented via Parking Status, we will start with the Models layer.

If we take our domain models that were defined earlier ('Lot', 'LotStatusSchedule', 'LotStatusScheduleDate', 'Status', 'StatusEvent' , 'StatusEventDate', and 'AdminUser') we are essentially converting each of those abstract domain models into practical Java classes. Instead of following the exact same design pattern as the database, we instead will be coupling necessary models together. So, a 'Lot' class has an instance of a 'LotStatusSchedule' class. A 'LotStatusSchedule' class has an ArrayList of 'LotStatusScheduleDate' instances. A 'Status' class stands alone and is not coupled to any other object. A 'StatusEvent' class has an ArrayList of 'StatusEventDate' instances. Essentially, we will be using all of these Model classes throughout the rest of our Spring Boot backend.

The next layer that Parking Status implements in the MVC is a DataAccessService layer. This layer can be considered 'the primary contact' between the backend and the actual data in the database. Each major Parking Status object ('Lot', 'Status', 'StatusEvent', and 'AdminUser') has its own DataAccessService class where it deals with queries to that database based on the operations desired. Each of these classes utilizes dependency injection[7] via an annotation ('@Autowired') in its constructor. The benefit of dependency injection is that instead of creating a 'new' object of whatever is referenced, it will instead access the current one in memory if it exists, allowing it to be resource conscious. Each DataAccessService class has an iteration of the following methods: 'selectAll' which queries all

[6] "Spring vs. Spring Boot vs. Spring MVC" *JavaT*point, https://www.javatpoint.com/spring-vs-spring-boot-vs-spring-mvc#:~:text=Spring%20Boot%20is%20a%20module%20of%20Spring%20for%20packaging%20the,to%20build%20Spring%2Dpowered%20framework.

[7] Karia, Bhavya. "A quick intro to Dependency Injection: what it is, and when to use it" *Free Code Camp, Inc.*

of a given object and returns a List of said object, ‘selectById’ which takes in an id variable in order to perform a query to find the object that matches that id and returns the respective object if it is found, ‘insert’ which takes in an object model from our Models layer and performs a new query that creates a new entry in the database of that model’s attributes, ‘update’ which takes in an id of the desired object as well as a new instance of the model with the newly updated attributes and performs an update query on that entry in the database, and ‘remove’ which takes in an id and performs a delete query on the database for the entry that corresponds to that given id. All four DataAccessService classes have those standard methods. Some of those classes require additional methods to allow for a cascading update or delete based on the object that is being handled. All of the DataAccessService classes are annotated with ‘@Repository’ to indicate that it is a data source that Spring Boot must utilize dependency injection with. Lastly, each domain object has a corresponding class called an InsertDataMapper that properly formats the desired objects to be queried into a format that PostgreSQL will accept.

The next layer that ParkingStatus implements is that of the Service layer. Once again, each of the main four Parking Status objects has their own Service layer class. Each one has an annotation of ‘@Service’ about the class declaration, an injected instance of the corresponding DataAccessService class via ‘@Autowired’, and an injected constructor. Essentially, functions in this layer correspond to those in the DataAccessService layer and generally match in terms of parameters, return types, and signatures. Each function makes a call to its injected DataAccessService class’s corresponding function. This layering allows for there to be clarity and a disconnect between the business logic and actual data within the datasource.

Last but not least, the final MVC layer for Parking Status is the Controller layer. When dealing with REST API creation, this is the layer that allows for CRUD operations to be

initialized and sent to the other layers. Once again, each of the four main ParkingStatus objects has its own Controller class. Each controller class creates its own endpoint (each named as '{urlname}/api/v1/{objectname}'. This helps designate what each endpoint represents. The 'api' section is self-explanatory as it explains that this is an API endpoint, 'v1' is a labeling practice that I was taught during my time at CGI Inc., whereas new versions of the API are created and implemented, instead of replacing the existing API, we would create a new endpoint with the next subsequent number (so in this case the next would be 'v2'). This allows us to have a versioning history of our endpoints. Lastly, the end of the endpoint is simply the name of the object that we are accessing ('lots', 'status','statusevents', or 'adminusers').

Each Controller class is annotated with a '@CrossOrigin((and the url that the frontend is hosted on))' in order for requests to be made from the frontend to the backend, a '@RestController' to indicate that this is a REST API endpoint, and a '@RequestMapping((with the url practice that we described earlier.))'. Each Controller class also has an injected corresponding Service class and is used throughout each of the methods. Methods from that Service class are called accordingly. Once again, these functions mimic those of the Service layer, however,each function is annotated with either a 'Get', 'Post', or 'Delete' mapping to indicate which type of REST API request is required to trigger that function throughout the backend. The return values of each function in the Controller are included in the response body of the API request and are used to conditionally render aspects on the frontend. This allows us to handle successful and unsuccessful requests properly on the frontend. Each object is also represented in a JSON format on response and request bodies where each field is marked as a new attribute within the JSON object.

The main tools that I used for creating the backend were the Intellij IDE (Community Edition 2021) for handling Spring Boot and Postman for quickly testing out the APIs that Parking Status implements.

Frontend

The final important feature of Parking Status is the capability to facilitate user interaction for non-admin and admin users. There needed to be a UI that would allow for a straightforward interaction with the data so that it could be altered and presented. The frontend is built using the popular JavaScript frontend framework (or library) known as React. Created by Facebook in 2013, React is essentially constructed using pieces called Components. Components are just JavaScript functions that return a special version of HTML known as JavaScript XML (JSX). JSX consists of JavaScript code with HTML markup; allowing for JavaScript functions and variables to be directly accessed within the markup. The large appeal with React is with this puzzle-piece-like nature of its Components as they can easily be altered and edited to suit certain needs, stacked on top of eachother to render certain parts of the UI, and be part of a 'Component tree' where Parent Components can house Child Components[8]. All that it takes for a Component to be used in another one is that it is imported into the other Component. A React Component is typically created either with an opening and closing tag ('<ComponentName></ComponentName>') *or* as one singular tag ('<ComponentName/>'). Other aspects of a React project are a single 'root' Component named 'App.js' where all other child Components can stem from, a singular 'index.js' file where the main 'App.js' components

[8] "React in 100 Seconds" *YouTube* , uploaded by Fireship, 8 Sep 2020, https://www.youtube.com/watch?v=Tn6-PIqc4UM&list=PL0vfts4VzfNgUUEtEjxDVfh4iocVR3qIb

are rendered, various index '.html' and '.css' files, a package.json file, and a node_modules folder where all of the stored Node Package Manager (Npm) packages are stored.

Another appeal of React is the ability to run the server and make changes to the source code that the server will *react* to by re-rendering the presented components again.

React has a few important aspects that Parking Status explicitly utilized in order to implement the UI in a functional manner such as the React-router-dom package, a specific parameter that can be passed to each Component (typically referred to as a 'prop' or 'props'), states, and 'lifecycle' methods (often referred to as React Hooks).

The React-router-dom package allows for premade React components to be utilized so that certain Components can be routed to a different URL (Route), rendered to certain URLs *exclusively*[9] (Switch), and kept in sync with the URL (BrowserRouter). This simply makes routing between components easier on the frontend.

Props, or properties, are parameters for a given component. The way a prop is passed to a component is by sending a variable name to the component with a value ('<ComponentName intProperty={3005}>'). The way that property is then accessed inside of the Component is by declaring a variable to be received as a parameter in the Component function ('function ComponentName(props){}'). In order to reference the passed in property, one must access the value passed in from the props variable ('props.intProperty'). Along with variables, there is the possibility to pass in functions as props as well. This technique is especially useful for when one wants to update a parent component based on a condition of a child component.

As a person experienced in the backend, States are an aspect of React that were odd to deal with at first. States are somewhat comparable to the likes of fields that a particular Component stores. However they are dissimilar in that, in order to update a State field, one must

[9] "React Router" *React Training*, https://v5.reactrouter.com/web/api

call a function called ‘setState’ to do so if one is using class components. Assigning a State value directly does not work. However, Parking Status uses mostly functional Components, requiring us to have to handle States in a different manner since functions cannot necessarily store fields. This brings us to React Hooks/Lifecycle methods.

A React Hook is essentially a method that allows us to handle the state of a Component without needing the Component to be a class. React Hooks are also used for triggering certain render events once certain values change or if one wants to access a function from a separate JavasScript function or other mounting[10] in order to make an API call. The main three React Hooks that Parking Status used were ‘useState’, ‘useEffect’, and ‘useContext’. ‘UseState’ requires a declaration of a ‘const’ data type with a variable name and a setter method name defined and it must be initialized to a default value. Upon editing the value, the setter method is called with the newly desired value being passed in. ‘UseEffect’ requires a function be passed to it as well as an array with 0 or more variables being passed within that array. Depending on what variables are passed in, upon the change of those values within the Component, the ‘useEffect’ function will run. If no variable is passed into the array, then ‘useEffect’ runs on the initial render and then never again until the page is re-rendered. ‘UseContext/CreateContext’ essentially allows for global variable values that can be accessed across the entire React application. Essentially, one creates a new React component , calls the CreateContext hook that initializes a value, and then that variable can be exported to a different Component. In order to update this value, the Component that stores the context must be imported and one must call the ‘useContext’ Hook and pass in the imported Component. The Child Component that one wishes to send the value to must be wrapped in a special component called a provider and given a value (‘<ContextComponent.Provider

[10] “React Router”

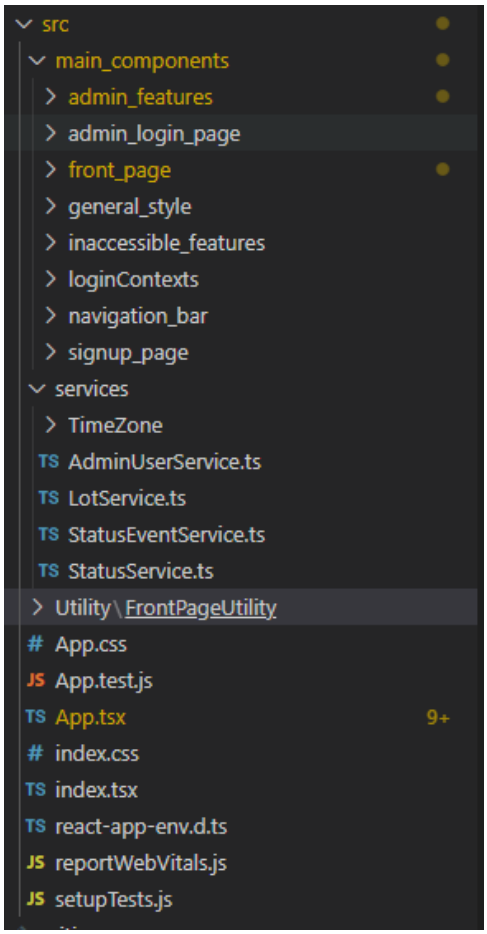
value={2013}><ChildComponent/></ContextComponent.Provider>'). These React Hooks are essentially the 'engines' behind much of the functionality within the UI Components of Parking Status.

- ➢ The Benefits of TypeScript
    - One aspect about React that I have refrained from mentioning until now is the usage of TypeScript within the application. TypeScript was released in 2012 by Microsoft and was the 2nd most loved technology on the 2020 StackOverflow Survey[11].
    - What exactly is TypeScript? In technical terms, TypeScript is a superset of Javascript. In plain terms, TypeScript offers options beyond that of regular JavaScript. The reason that TypeScript has been soaring in popularity as of late is due to its ability to add dynamic type-checking to JavaScript variables and functions, something that JavaScript severely lacks and can result in more bugs not being caught until displayed in the browser. As a simple example, instead of declaring a variable as 'var name = "name";' TypeScript forces us to apply a particular type annotation to the variable 'var name : string = "name";'. This simple feature stops the developers from 'shooting themselves in the foot' by accidentally performing operations to variables that cannot not occur (such as adding an int to a String, for example). This type checking can be overridden by annotating a variable or function as an 'any' type, however, it completely defeats the purpose of using TypeScript. Another option is that a developer can define their own types and then annotate variables to them as well.

[11] "How to use TypeScript with React... But should you?" *YouTube*, uploaded by Fireship 19 Apr 2021,https://www.youtube.com/watch?v=ydkQlJhodio&list=PLLTzpdwgv_AeMMvCci1cmvW9N2Wp3VH-v&index=9

- The frontend of Parking Status uses TypeScript for the React files and Components rather than regular JavaScript.
- What does this mean for React? There are only a few matters that now change due to this. 1) Instead of having React Component files with '.js' extensions , the files will now have '.tsx' extensions to indicate a TypeScript version of the JSX that we are using. 2) We will be having a 'tsconfig.json' file that will communicate with the compiler and convert any TypeScript files to JavaScript so that it can be consumed by the browser (TypeScript code cannot be run in the browser). 3) All React components must now also be annotated in the same nature as variables. For the purposes of Parking Status, each Component now has a pre-built 'React.FC' (Functional Component) type that can now be applied to the entire Component.

The Parking Status folder structure of the React frontend is as follows.



- The '**src**' folder is the parent directory that contains the bulk of the React Application (App.tsx, stylesheets, etc.)
- The '**main_components**' folder houses:
  - *admin_features*: All UI components necessary for performing CRUD operations on 'Lot', 'Status', 'StatusEvent', and designated timezone objects.
  - *admin_login_page*: the admin user login page.
  - *front_page*: the LotAccordion display that conditionally changes based on the time and scheduled dates, the ColorLegend, and a Context Component for the current time of the designated time zone.
  - *general_style*: stylesheets that are used across the entire React application
  - *inaccessible_features*: a default component that renders when a non-admin user goes to an admin-only route and informs the user that they do not have access.

    - *loginContexts*: Context components for storing the existing admin login status of the application.
    - *navigation_bar*: Navigation Bar Components and stylesheets.
    - *signup_page*: Component for signing up a new admin user.
- The '**services'** folder houses regular TypeScript files that deal with either the API endpoints that our backend serves or handles requests dealing with the WorldTimeAPI in regards to the designated time zone and current time. These service classes are implemented across the React Components using the 'useEffect' Hook and a third party library called 'axios' that is designed for handling API requests.
- The '**Utility'** folder houses 'FrontPageHandler.ts'. This is a regular TypeScript file for discerning which 'LotStatusScheduleDates' and/or 'StatusEventDates' should be marked as active on a given lot based on the current time.
- All Routing purposes were defined within the '**App.tsx**' file.

All of these pieces (TSX Components, TypeScript classes, and calls to various APIs) are what shape the user experience of Parking Status.

Upon a failed creation of an object or failed signup of a new admin user, an error message will be displayed to indicate why the submission was unsuccessful and how to fix it.

In terms of tools that I used when developing the frontend, I used Visual Studio Code with an integrated Bash Terminal to run 'npm start' for running the frontend server, a Google Chrome browser, and the official Facebook React Developer Tools Google Chrome Extension for quick breakdowns of Component Trees[12]. In terms of additional npm packages that I utilized, I used 'axios' for handling API requests[13], the 'typescript' dependency for making the application compatible with TypeScript, and 'react-router-dom' for handling routing. In terms of third-party technologies, I used the WorldTimeAPI to gather a list of all existing time zones and get current date data for a given timezone[14]. I also used the 'browser-hash.js' library for handling user-side hashes of password attempts for admin users trying to login[15].

---

[12] "React Developer Tools" *Chrome Web Store*, Offered by Facebook, https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi?hl=en

[13] "Axios" *Axios Media*, https://www.axios.com/

[14] "WorldTimeAPI" *WorldTimeAPI*, https://worldtimeapi.org/

[15] "How to generate a SHA-256 hash with JavaScript" *Medium*, written by remarkablemark, 29 Aug 2021, https://remarkablemark.medium.com/how-to-generate-a-sha-256-hash-with-javascript-d3b2696382fd

Overall, the frontend of the solution is what makes interacting with the Parking Status data manageable.

My Development Process

Attempted Process

My initial goal for designing and developing Parking Status is a far cry from *how* it was actually implemented in some places. In my initial proposal I stated, "...I want to also see if the Agile Methodology that I was exposed to in one of my past internships would also help me when building this solution.". I created an Agile board on Jira and even initial user stories[16]. However, I quickly realized why Agile methodologies were used for software development *teams* rather than individuals[17]. Their main appeal is to communicate product features and issues across the entire team which may comprise different roles such as frontend developers, backend developers, quality assurance, technical leads, Scrum masters, and Project Managers. Each of these roles is vastly different in its duties and responsibilities, but when the only person on the development team is *you*, there seems to be less of a need to communicate with yourself on requirements and scope.

That is not to say there should be *no* communication. I needed to leave documentation all across different files in order for me to know what was going on later and with the hopes that it would be understandable for any others who may be looking at my implementation. Any time that I found an online solution to a problem, I made the effort of marking down the link to that solution within the areas of the code where it was implemented in. I also attempted to place descriptions with chunks of code that demonstrated their purpose.

---

[16] "Jira Software" *Atlassian*, https://www.atlassian.com/software/jira
[17] "WHAT IS AGILE? WHAT IS SCRUM?" *CPrime, Inc.*, https://www.cprime.com/resources/what-is-agile-what-is-scrum/

In terms of Minimum Viable Product (MVP) and scope, instead of a team needing to recognize their limitations and abilities to deliver on a solution, I did have to realize my limits in order to deliver the solution on time. For me, my limitations rested in a few of the technologies that I tried to use. I initially attempted to use the .NET Core framework for Parking Status's backend. I was planning on implementing an Onion Architecture that would hopefully allow for a strong domain-driven design[18]. However, after following a tutorial online and attempting to branch out into creating an endpoint for the data, I quickly realized my personal limits with this technology[19]. It came to my attention that it would not be feasible to build such a vital part of the application with a technology that I was barely familiar with. I had internship experience in building an API that *connected* to a backend in .NET (5 not Core), but between the specifics of that and my unfamiliarity with 'C#' for such a large task, I decided I needed to quickly switch to something I was more familiar with. This brought me to the Spring Boot framework, something that I understood from the top-down in implementation and used in my native programming language of Java. I also initially intended to use Microsoft SQL Server as the database for Parking Status, however when attempting to connect the Spring Boot application, I ran into many difficulties and was advised to use PostgreSQL as the database instead. Lastly, while I was not fully familiar with React, I was able to take advantage of my minimal experience of it from side projects, previous internship examples where my peers worked with it, and other forms of help in order to make a frontend that was at least functional and intuitive, if not pretty and glamorous.

[18] Pal, Tapas. "Understanding Onion Architecture" *Codeguru*
[19] "Onion Architecture in ASP.NET Core" *CodeMaze*, https://code-maze.com/onion-architecture-in-aspnetcore/#infrastructure-layer

Actual Process

My initial technology stack consisted of .NET Core, Microsoft SQL Server, and React with Typescript and a handling of the stack in an Agile manner. The true technology stack of Parking Status became Spring Boot (Maven), PostgreSQL Server, and React with TypeScript and it was implemented more so in a Waterfall approach[20] where I simply implemented one feature after another in a very linear fashion. I started out constructing the abstract Domain Models and database schema, so when I realized I needed to switch backend frameworks and database servers, the transition was straightforward. I built about 60% of the Spring Boot backend before constructing the database that would need to be connected to that backend. During my process of setting up the endpoints, I heavily relied on Postman to test out request options to handle the CRUD operations. After I considered the backend to be fully done, I then moved on to the frontend.

As a developer who has spent most of his time on the backend of projects, learning to build the frontend in React was a daunting but ultimately rewarding task to take on. With the usage of features such as lifecycle methods, props, and state management, I feel as though those aspects heavily went against the grain of regular programming paradigms. It took a fair amount of time and help for me to learn how React handles its assets when creating a responsive UI. Adding TypeScript, while helpful for catching bugs and definitely a tool I enjoyed using, did not make the initial process easier. I would say that I was able to roughly construct the backend in about two to two and a half weeks. The rest of the time was spent trying to understand and implement the frontend, which absolutely threw me out of my comfort zone. In terms of MVP, I

[20] "What is the Waterfall methodology?" *Adobe*, https://www.workfront.com/project-management/methodologies/waterfall

eventually settled on a UI design that , while not the most visually appealing to look at, could hopefully be understood relatively quickly and easy to use.

Lastly, in terms of version control, I used Git and have Parking Status hosted on a public GitHub repository. For handling the backend, I used Intellij's built-in Git UI to make commits and pull any new changes. For handling the frontend, I used VS Code's option to integrate Git Bash into its environment. I used Git Bash to run 'npm' and Git commands.

Documentation

Other than this very document, the only other documentation is that of the database schema and a Configuration Guide that explains how to set up Parking Status locally on your machine. The main reason for a Configuration Guide is to mostly ensure that any additional libraries on the frontend or backend are properly installed as well as to serve as a step-by-step guide on how to construct the PostgreSQL database and its credentials using the SQL scripts found in the repository.

Future State

While I was able to implement most of what I sought out to in Parking Status: 1) the ability to create, edit, and delete Lots, Statuses, and StatusEvents, 2) a clear homepage that dynamically changes based on the assigned time zone, current time, and date to communicate to a non-admin user, 3) a clear UI that allowed for easy interaction with the data, and 4) a simple admin login and logout system, there are many possible features that I think would improve upon the current Parking Status system, given more time and experience with certain technologies.

Additional Features

I originally planned for 'Lot', 'Status', and 'Status Event' objects to be able to store images. However, after learning that storing images in a database is not as straightforward as desired and yet would be *another* local machine concern as images would need to be stored in local memory, I decided that this would be beyond MVP for me during the initial development process. The intention was to allow for a single upload of an image within the three main objects for whatever purposes the admin decided to use them for. Some simple use cases were that of a picture of the lot, a marked map indicating the location of the lot on campus , and a logo or icon to indicate the Status or StatusEvent. While there are fields in both the domain models and database to store images and names, they are not used.

In addition to creating a simple login system, a slightly more robust admin login system should be implemented if this were to be officially used. The main flaw in this system is that any user can 'Sign Up' with their email and become an admin. I considered making the 'Sign Up' feature itself something that would be an admin-only feature, however, that would require manually adding a default admin user into the database in order to access it. If we go further, this would basically require a large initialization process for the first time the solution would be ran and in terms of MVP, that was not feasible with my given skill set or time allotted.

My original intention was to implement a Google Calendar-like UI component for scheduling LotStatusScheduleDates and StatusEventDates with the feature of easily dragging and dropping items in different areas of a weekly or yearly calendar. I found pre-made React components with the capability to do this such as 'react-big-calendar'[21] and 'react-datepicker'[22]. However, the default implementations of these revolved around a yearly calendar system where

[21] "react-big-calendar" *Jason Quense (jquense)*, https://github.com/jquense/react-big-calendar
[22] "React-Date-Picker" *Wojciech Maj (wojtekmaj)*, https://github.com/wojtekmaj/react-date-picker

each day entry represents a specific date on the calendar. For my purposes, converting that to a generic seven-day week for a ‘lot status schedule’ was not clear within either of the documentations. In this case, I decided to go with creating my own day and date selection for the LotStatusScheduleDates and StatusEventDates.

The UI itself is not the best-looking UI ever created. Since creating the frontend required me to embrace React and frontend development, I still have much to learn in order to make enticing UIs. A feature to add more fluid animations and styling options would allow Parking Status to look more visually appealing to any end user. While any failed attempt for logging in or object submission is communicated to the end user, animations for successful attempts would have been a good place to start.

The front web page was intended to conditionally render every time the time changes. However, with the usage of the WorldTimeAPI, this would have been extremely resource-intensive: both to my backend as well as the WorldTimeAPI itself. The WorldTimeAPI gets the current time down to the second and making an API request at the change of each second would have caused the requests to eventually ‘time out’ as the API would consider a large number of requests within a certain amount of time as a potential security threat. I could have attempted to detect *every* minute change, but that also seemed resource intensive and would require me to be slightly more comfortable with React Hooks. To remedy this, every Lot entry on the homepage that has an active status has an entry labeled as ‘Duration’ that shows the start and end time of that status. Refreshing the homepage allows for the current date information to be updated. A future implementation of this would allow for an efficient way to detect time changes and render those changes without abusing any data sources.

In terms of dates being scheduled for certain times, while there is an organization of that data, it would be ideal if the admin user was able to receive popup notifications on potential time conflicts whenever certain aspects of the data were altered. For example, if the admin wants to schedule a 'Status Event' that conflicts with a parking lot's schedule, notifying the user before confirming the data change would be a better design. Overall, other than checking if there are conflicting ids within the system, conflict handling was not implemented. Personally, I think this feature was simply too out of Scope during my initial development process.

Lastly, a mobile version of this web application would serve as an even more convenient way in which guests and visitors could quickly see the parking status of a certain area. In order to implement this, I would have needed to learn how to use React's mobile extension React Native[23]. Technically, accessing the link that Parking Status would be hosted on would be allowed and the UI components render and fit accordingly. However, making a mobile-specific version of Parking Status would have been a more thorough design.

Implementation

There are many coding aspects of this solution that are not as efficient or written as well as they could have been.

On the backend, the DataAccessService classes each have about 400 lines of code. Looking at the DataAccessService and trying to discern what is happening may be difficult for a new set of eyes. In a future example of Parking Status, I would create a PostgreSQL query class with the sole purpose of conditionally creating SQL queries that satisfied the PostgreSQL syntax. I did wrap each domain model that was meant to be queried to the database into its own

[23] "React Native in 100 Seconds" *YouTube*, uploaded by Fireship, 28 Sep 2021, https://www.youtube.com/watch?v=gvkqT_Uoahw

InsertDataMapper class, however, this can seem ‘clunky’ and more ‘complex than necessary’ at times.

On the frontend, I did not fully take advantage of the capabilities of TypeScript when it came to type-checking. When dealing with any non-primitive data type or React component, I usually annotated the value with an ‘any’ type which can defeat the purpose of using TypeScript. I mainly did this due to the time consumption of creating custom types that would need to span across the entire React application. A thorough redefining of ‘any’ type variables into their own custom types would allow for a product that is closer to industry standard. Also, since I am fairly new to React, there are many instances where I was not as efficient as an experienced React developer could be. As an example, for the object ‘create’ and ‘edit’ components, much of that functionality and code is similar, so I could have just made one file for each rather than a ‘create’ and ‘edit’. However, this would have required overhead on sending in a properties value that I would have not been familiar with at the time.

Faith Integration

In reflection upon ways in which Parking Status demonstrates my faith as a follower of Christ, I have settled on two main factors that I think demonstrate that: communication and intentional design.

Communication

Matthew 18:15-17 states, “If your brother sins against you, go and tell him his fault, between you and him alone. If he listens to you, you have gained your brother. But if he does not listen, take one or two others along with you, that every charge may be established by the

evidence of two or three witnesses. If he refuses to listen to them, tell it to the church, let him be to you as a Gentile and a tax collector."[24]. While accidentally parking in the wrong place is not a sin directly and the institution that allows for parking may not be a church per se, I think there are elements within this quote that demonstrate the value of clearly communicating the quarrels and issues between two or more parties. In the case of parking, that could be a person parking in the wrong spot or a visitor confused by the unfamiliar parking norms of the institution. Parking Status's attempt to provide a platform and system of clear communication on the simple, yet sometimes frustrating issues in a broken world is an effort to allow for clearer communication between people with the hopes of avoiding dramatic events such as parking tickets or visitor confusion.

Intentional Design

In *Shaping A Digital World*, Derek C. Schuurmann makes the point that technology is not neutral, "The concept of technological volition recognizes that technology is shaped by human will...the fact is that technology is value-laden."[25]. Essentially, the designer of a given technology *will* demonstrate their values in what it is that he or she creates. In any discussion about general technology, there is the assumption that the discussion pertains *only* to digital technology: software, mobile devices, data collections, etc. Merriam-Webster defines the term 'technology' as a) 'the practical application of knowledge especially in a particular area' and b) 'a capability given by the practical application of knowledge'. When discussing existing parking systems, we must also understand the ways in which it uses technology in its design (eg. street signs, digital or physical maps, marked sections of pavement, sections marked off by cones, etc.). With the

[24] Matthew 18:16 (English Standard Version) *Crossway*
[25] Schuurmann, Derek C. *Shaping a Digital World*, p.14-15 , InterVarsity Press ,2011

reminder that technology is more than merely digital, we can now recognize that something such as a parking system within a city or campus is also value-laden. Many parking systems are, in fact, designed to be ambiguous with the hopes of ticketing unknowing commuters. Is that a good value that is embedded within that system? There is the argument that ticket payments will help the institution that is charging the commuter (eg. a public area or a college campus), which is a fair and true argument. However this argument brushes aside any flaws concerning the poor and confusing communication that certain parking systems may have as well as the question of 'Am I loving my ticketed neighbor?' with the poor design. Schuurmann also states, "Computer scientists and engineers (or in my case, any system designer) should seek guidance from domain experts during the design process. These experts help ensure that design decisions are not driven primarily by technical considerations but rather with the primary purpose of the end user in mind."[26]. Parking Status is an attempt at providing a better parking system design for notifying commuters. As suggested by the future state (see 'Future State' section), this implementation is not perfect nor the most optimal for the designer or end user. However, the embedded values of design for Parking Status were meant for valuing the end users (admin and non-admin) as fellow neighbors.

What I Learned

On a surface level, developing and designing Parking Status allowed me to use skills that I had gained from previous programming courses, previous internship experience, and side-project work. I was able to sharpen and enhance my skills as a Java Spring Boot developer and database administrator. I was able to learn my first official frontend framework with React and address an area of my expertise within software development that I believe was lacking. I

[26] Schuurman, p.78

was able to harmonize a backend, frontend, and database in order to create my first full stack application that tackled a real-world problem. In terms of conceptual ideas that I learned from developing Parking Status, I believe I learned: 'how to not always accept "no" for an answer' and 'how to sometimes accept "not yet" as an answer'.

How to Not Always Accept "No" for An Answer

During the initial capstone project selection process, I had recognized this ambiguity of parking within Covenant College's campus: leading me to want to develop Parking Status. There were many instances where parking situations on campus could have been communicated better by the Safety & Security department such as: my time as a prospective student and previewer before becoming an official Covenant student (there was great confusion with my ride on where to best park since resources about this were not clearly displayed on the college website), countless Safety & Security emails about new parking situations due to events that people visiting Covenant would not be informed about, and times where peers have gotten ticketed due to no knowing the current nature of a parking situation. I originally pitched this idea to the Technology Services department, however, there was belief that this data would not be updated enough or considered a 'great fit' to warrant a project solution. Instead of accepting the 'no' from this response, I considered my environment where manual emails were still constantly being sent out, temporary parking maps were poorly illustrated, and parking notices were issued to certain guest vehicles and decided that a) there was actually a need for this solution in my current surrounding and b) even if there was still a case to be made for there *not* to be an implementation of this solution, there are many other institutions and locations that could use this in a practical and mindful manner. While I intentionally meant to make this specific to the

Covenant College campus, given that it is not considered a ‘great fit’ of a solution for the campus to certain parties, this allowed me to make this a dynamic and highly customizable solution to the needs of any institution or location that wishes to use it.

How to Sometimes Accept “Not Yet” as an Answer

Whether it be the need to switch out backend technologies due to my lack of experience, the inability to implement an Agile methodology by myself, the items listed in the ‘Future State’ section, or initial lack of knowledge of React, JavaScript, TypeScript, or frontend development in general, the development of Parking Status taught me to appreciate patience when learning a new technology and recognize my current limits amidst a deadline. Learning how to develop a frontend was extremely rewarding as a mostly backend developer. Being forced to make the timely decision to switch from .NET Core to Spring Boot and from an Onion Architecture to MVC also taught me the importance in ‘making a decision’ while a deadline is approaching rather than ‘being a sitting duck’. When considering the tech industry, I think being able to value one’s current limit on deliverables (whether it be a language, framework, methodology, tool, data organization, or requirement) is a valuable skill to have in order for things to at least ‘get done’. Regardless of the implementation, technology is always changing and systems are always being rewritten, so, it is unrealistic to think that the first time a solution is implemented is the best implementation. As a follower of Christ, I specifically see an opportunity to practice humility and recognize one's human finitude within the ability to recognize limitations. Once the deadline is completed, one can always go back and learn the technologies in a less intense and lower-stakes environment and I plan to do the same with .NET Core, Onion Architecture, React, React Native, and TypeScript.

Acknowledgements

During my time developing Parking Status between August 2021 and December 2021, I used three individuals as resources to help me with certain aspects of Parking Status: Julian Laury, Evan Moses, and Jake Beaton.

I would like to thank Julian Laury for answering any React questions related to necessary APIs, lifecycle methods, and component organizations as well as Typescript specifics.

I would like to thank Evan Moses for answering any Spring Boot concerns I had during the process of developing the backend, helping solve a JSON response body issue that made handling the backend data much easier, and consulting me on admin user login authentication protocols .

I would like to thank Jake Beaton for walking me through the experience of building an API in .NET 5 from June 2021 to July 2021 at CGI Inc., the many explanations behind an Onion Architecture, and the explained importance behind API versioning.

Sources

Developmental
(sources that provided coding solutions for problems)

Branchfield, Seann. “How To Pass Props to Components Inside React Router” *Medium*

Chavan, Yogesh. “How to Build an Accordion Menu in React from Scratch – No External Libraries Required” *Free Code Camp, Inc.*

“Onion Architecture in ASP.NET Core” *CodeMaze*, https://code-maze.com/onion-architecture-in-aspnetcore/#infrastructure-layer

“3 ways to display two divs side by side (float, flexbox, CSS grid)” *coder coder!*, https://coder-coder.com/display-divs-side-by-side/

Dietrich, James. “How to Refresh a Page or Component in React” *Upmostly*

“Tic Tac Toe” *Facebook Inc.*, https://codepen.io/gaearon/pen/gWWZgR?editors=0010

García, Ceci García. “Implementing private routes with React Router and Hooks” *Medium*

“Returning Multiple values in Java” *GeeksforGeeks*, https://www.geeksforgeeks.org/returning-multiple-values-in-java/

Ha Minh, Nam. “Spring Boot Connect to PostgreSQL Database Examples” *CodeJava*

Jain, Viral. “Create Database Project In Visual Studio 2015” *C# Corner*

“How to generate a SHA-256 hash with JavaScript” *Medium*, written by remarkablemark, 29 Aug 2021,https://remarkablemark.medium.com/how-to-generate-a-sha-256-hash-with-javascript-d3b2696382fd

MacLean, Lisa. “Domain Model vs. Data Model” *Study.com*

Murobayashi, Chad. “Create a Month, Week, and Day View Calendar with React and FullCalendar” *gitconnected*

“React Router” *React Training*, https://v5.reactrouter.com/web/api

Pal, Tapas. “Understanding Onion Architecture” *Codeguru*

“Tutorial (ASP.NET Core)” *Present Facebook Inc.*, https://reactjs.net/tutorials/aspnetcore

“Day Name from Date in JS” *StackOverflow*, asked by Casey Falk,, 28 Jul 2014, https://stackoverflow.com/questions/24998624/day-name-from-date-in-js

“Basic React form submit refreshes entire page” *StackOverflow*, asked by Mike Smith Alexiss, 21 Jun 2021, https://stackoverflow.com/questions/50193227/basic-react-form-submit-refreshes-entire-page

“How to change the table name in visual studio 2013 in design mode?” *StackOverflow*, asked by Samiey Mehdi, 12 Dec 2013, https://stackoverflow.com/questions/1799802/dropping-a-table-with-visual-studio-2010-database-project#

Welch, AJ. “How to List Databases and Tables in PostgreSQL Using psql” *Chartio*

“Typing Form Events [React + TypeScript]” *YouTube*, uploaded by Atila IO, 24 Mar 2021, https://www.youtube.com/watch?v=BYsQE3Nh9IE&list=PLLTzpdwgv_AeMMvCci1cmvW9N2Wp3VH-v&index=31&t=566s

“How to Conditionally Render Components in React.js - Part 7” *YouTube*, uploaded by Ben Awad, 17 Sep 2018, https://www.youtube.com/watch?v=3wvdq_j5S1c&list=PLLTzpdwgv_AeMMvCci1cmvW9N2Wp3VH-v&index=4

“Fetching Data from an API with React Hooks useEffect” *YouTube*, uploaded by Ben Awad, 28 Oct 2018, https://www.youtube.com/watch?v=k0WnY0Hqe5c&list=PLLTzpdwgv_AeMMvCci1cmvW9N2Wp3VH-v&index=6&t=315s

“React Hooks useContext Tutorial (Storing a User)” *YouTube*, uploaded by Ben Awad, 29 Jun 2019, https://www.youtube.com/watch?v=lhMKvyLRWo0&list=PLLTzpdwgv_AeMMvCci1cmvW9N2Wp3VH-v&index=11

"React Typescript Tutorial" *YouTube*, uploaded by Ben Awad, 5 Sep 2019, https://www.youtube.com/watch?v=Z5iWr6Srsj8&list=PLLTzpdwgv_AeMMvCci1cmvW9N2Wp3VH-v&index=26&t=885s

"How to Use ASYNC Functions in React Hooks Tutorial - (UseEffect + Axios)" *YouTube*, uploaded by Clever Programmer, 25 Jul 2020, https://www.youtube.com/watch?v=IbHuwpPwfoc&list=PLLTzpdwgv_AeMMvCci1cmvW9N2Wp3VH-v&index=39

"React TypeScript Tutorial - 10 - useState Future Value" *YouTube*, uploaded by Codevolution, 12 Sep 2021, https://www.youtube.com/watch?v=LNpWuRUIR5A&list=PLLTzpdwgv_AeMMvCci1cmvW9N2Wp3VH-v&index=30&t=296s

"Common React Mistakes: useEffect - Part 1" *YouTube*, uploaded by Jack Herrington, 31 Ma2 2021, https://www.youtube.com/watch?v=lStfMBiWROQ&list=PLLTzpdwgv_AeMMvCci1cmvW9N2Wp3VH-v&index=4

"How to Use ASYNC Functions in React Hooks Tutorial - (UseEffect + Axios)" *YouTube*, uploaded by Clever Programmer, 25 Jul 2020, https://www.youtube.com/watch?v=IbHuwpPwfoc&list=PLLTzpdwgv_AeMMvCci1cmvW9N2Wp3VH-v&index=39

"3 ways to run Spring Boot apps from command line - Java Brains" *YouTube*, uploaded by Java Brains, 10 Jan 2020, https://www.youtube.com/watch?v=Le5YjYNYtZg

"React JS + Spring Boot REST API Example Tutorial" *YouTube*, uploaded by Java Guides, 4 Jul 2020, https://www.youtube.com/watch?v=5RA5NpxbioI&list=PLLTzpdwgv_AeMMvCci1cmvW9N2Wp3VH-v&index=30&t=2037s

"How to type React events with TypeScript" *YouTube*, uploaded by Maksim Ivanov, 19 May 2021, https://www.youtube.com/watch?v=3PtiSy8G-Cs&list=PLLTzpdwgv_AeMMvCci1cmvW9N2Wp3VH-v&index=26

"Spring + JdbcTemplate + Query multiple rows example | Spring JDBC tutorial" *YouTube*, uploaded by Ram N Java, 13 Feb 2019, https://www.youtube.com/watch?v=aWJH_0iV7Pg&list=PLLTzpdwgv_AeMMvCci1cmvW9N2Wp3VH-v&index=23&t=54s

"How To add Foreign keys on relational Database in Visual Studio" *YouTube*, uploaded by Tech Enthusiast, 2 2017, https://www.youtube.com/watch?v=FYEfO2y0Uc4

Informational

"What is the Waterfall methodology?" *Adobe*, https://www.workfront.com/project-management/methodologies/waterfall

"A Comparison Between Spring and Spring Boot" *Baeldung*, https://www.baeldung.com/spring-vs-spring-boot

"WHAT IS AGILE? WHAT IS SCRUM?" *CPrime, Inc.*, https://www.cprime.com/resources/what-is-agile-what-is-scrum/

Karia, Bhavya. "A quick intro to Dependency Injection: what it is, and when to use it" *Free Code Camp, Inc.*

"Spring vs. Spring Boot vs. Spring MVC" *JavaT*point, https://www.javatpoint.com/spring-vs-spring-boot-vs-spring-mvc#:~:text=Spring%20Boot%20is%20a%20module%20of%20Spring%20for%20packaging%20the.to%20build%20Spring%2Dpowered%20framework.

Lisa M. "Research Proposal: Parking Tickets" *Wonder*

"Domain Modeling" Scaled Agile, https://www.scaledagileframework.com/domain-modeling/

Schuurmann, Derek C. *Shaping a Digital World*, InterVarsity Press ,2011
"Why Spring?" *VMware, Inc.*, https://spring.io/why-spring
"React in 100 Seconds" *YouTube* , uploaded by Fireship, 8 Sep 2020, https://www.youtube.com/watch?v=Tn6-PIqc4UM&list=PL0vfts4VzfNqUUEtEjxDVfh4iocVR3qIb
"How to use TypeScript with React... But should you?" *YouTube*, uploaded by Fireship 19 Apr 2021,https://www.youtube.com/watch?v=ydkQlJhodio&list=PLLTzpdwgv_AeMMvCci1cmvW9N2Wp3VH-v&index=9
"React Native in 100 Seconds" *YouTube*, uploaded by Fireship, 28 Sep 2021, https://www.youtube.com/watch?v=gvkqT_Uoahw

Tools and Technologies Used

"Mockflow" *Mockflow*, https://www.mockflow.com/
"Jira Software" *Atlassian*, https://www.atlassian.com/software/jira
"Axios" *Axios Media*, https://www.axios.com/
"React Developer Tools" *Chrome Web Store*, Offered by Facebook, https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi?hl=en
"React" *Facebook Inc.*, https://reactjs.org/
"GitHub" *GitHub*, https://github.com/
"Guava: Google Core Libraries for Java" *Google*, https://github.com/google/guava
"react-big-calendar" *Jason Quense (jquense)*, https://github.com/jquense/react-big-calendar
"Intellij IDEA Community Edition 2021" *JetBrains s.r.o.*, https://www.jetbrains.com/idea/
"Diagrams.net" *JGraph Ltd.*, https://app.diagrams.net/
"Git" *Microsoft*, https://git-scm.com/
"SQL Server 2019" *Microsoft* https://www.microsoft.com/en-us/sql-server/sql-server-downloads
"SQL Server Management Studio version 18.10" *Microsoft* https://docs.microsoft.com/en-us/sql/ssms/download-sql-server-management-studio-ssms?view=sql-server-ver15
"TypeScript" *Microsoft*, https://www.typescriptlang.org/
"Visual Studio" *Microsoft* https://visualstudio.microsoft.com/
"Visual Studio Code" *Microsoft* *https://code.visualstudio.com/*
"JavaScript" *Oracle*, https://www.javascript.com/
"Java™ SE Development Kit 11.0.12 (JDK 11.0.12)" *Oracle*, https://www.oracle.com/java/technologies/javase/11-0-12-relnotes.html
"PostgreSQL Server" *The PostgreSQL Global Development Group*, https://www.postgresql.org/
"Postman" *Postman, Inc.*, https://www.postman.com/
"spring initializr" *VMware, Inc.*, https://start.spring.io/
"React-Date-Picker" *Wojciech Maj (wojtekmaj)*, https://github.com/wojtekmaj/react-date-picker
"WorldTimeAPI" *WorldTimeAPI*, https://worldtimeapi.org/

Miscellaneous

Matthew 18:16 (English Standard Version) *Crossway*