Visitation Application

Final Project Report

Final Requirements, Design, Implementation/Testing & Installation/Delivery

NC Parks and Recreation

CSC 492 Team 6

Alex Bowen

Allison Church

Mathew Deel

Shruti Jadhav

Quentin Sieredzki

North Carolina State University

Department of Computer Science

12/9/2022

**Executive Summary**

*Author: Quentin Sieredzki*

*Reviewers/Editors: Shruti Jadhav, Mathew Deel*

Our sponsor is the North Carolina Department of Parks and Recreation (NCDPR), who manage 41 state parks, natural areas, and recreation sites across North Carolina. NCDPR has an existing suite of web-based applications that they have created to help manage different aspects of the park management that are vital to the day-by-day functions of the department. However, these applications are outdated and were not created using modern technologies or practices. Because of this, it is very difficult to read, maintain, and upgrade the existing applications. NCDPR has thus decided to begin the process of updating their existing applications in order to make them more maintainable, and to use more modern design practices. As such, our project is a continuation of a previous project that has started to migrate to a newer stack by building a new calendar application.

Our team is specifically tasked with remaking the Visitation application. NCDPR has a variety of Internet of Things (IoT) visitation devices that track the number of people and cars that enter into a park and upload that data to a cloud database. This information is important because the park attendance data is used to learn more about the practices of visitors through metrics and reports in order to make better business decisions and receive more money from funding. The existing application requires park staff to manually upload this information despite the fact that the IoT devices have API connections that could be used to automate the system. This manual input is slow, and open to human errors. Additionally, the existing data visualizations are cumbersome and difficult to understand. Our team must recreate the existing functionality into the new application, while including automatic data uploading from the IoT devices, updating the visualization, and following modern practices.

The solution that our team has proposed is to use the Slim Framework for PHP to recreate the Visitation application. We will use modern technologies such as PHP 8, MariaDB 10, and React 17 to continue the process of updating the DPR applications. This application will exist within a Docker container, similar to the old application. The application will get the Visitation count through the usage of Ubidots, an IoT platform that already compiles the information from the IoT devices. We will also redesign the existing database to make it more useful for our own project and future applications, as the existing database is very redundant and difficult to use.

Currently, users must log in to authenticate themselves and are given different permissions based on the no access, base level, manager, admin, or super admin roles. Authenticated users can then view a list of devices, their recorded visits, and the counter rules each device uses to modify visits as well as add/edit/delete any data on these lists. The dashboard also displays a graphical representation of visits over time based on a park and a timerange. It also displays comparison metrics which give statistical data about changes in visitation over time. We also configured the Ubidots webhook functionality to automatically push data out when IoT devices collect new visits so that our database contains all archived and current data stored on the cloud server. The next steps would include updating the dashboard further with more comparison metrics as well as updating the unit testing to have better output and accommodate for error checking API calls.

**Project Description**

*Authors: Allison Church*

*Reviewer/Editor: Quentin Sieredzki*

## **Sponsor Background**

|  | **North Carolina Department of Parks and Recreation** |
| --- | --- |
| Address | Nature Research Center, 2nd Floor  121 W. Jones St.  Raleigh, NC 27601 |
| Phone | 919-707-9300 |
| Email | state.parks@ncparks.gov |
| Contacts | Eric Estes, *Deputy Director of Administration*  John Carter, *Database Manager*  Cathy Cooper, *Technical Support*  Cole Goodnight, *User Support Technician*  Joshua Roddy, *User Support Technician* |

*Table 1: Sponsor Contact Information*

The sponsor of our project is the North Carolina Division of Parks and Recreation (NCDPR), a division under the North Carolina Department of Natural and Cultural Resources. While the main office of NCDPR is located in Raleigh, NCDPR manages 41 state parks, natural areas, and recreation sites across North Carolina. The mission of NCDPR is to conserve North Carolina’s natural beauty, ecological features, and recreational and cultural resources, to provide and promote outdoor recreational opportunities, and to provide educational opportunities that promote stewardship of North Carolina’s heritage. To support the preservation of the state parks, natural, and recreational areas, NCDPR employees record the number of visitors at each of the parks on a daily basis. These visitation records serve to justify funding to the parks, determine the carrying capacity of visitors for each park, and understand park visitation trends, which can help NCDPR predict staffing needs for each park.

## **Problem Description**

NCDPR has a LAMP stack system that hosts multiple applications that park field staff use, including a legacy visitation application that park staff use to manually input data recorded by IoT counters. The process of manually inputting this data is tedious and can be subject to human error. The legacy visitation application is also written as a single-file, procedural application, which makes it difficult for the development team to maintain and upgrade the application. Moreover, the visitation application and other legacy applications are written in PHP5, a version that was declared to have reached its end of life by PHP developers by 2019 [1]. An unsupported software version is more vulnerable to cyber-attacks and may create incompatibilities when interacting with other software packages during development. NCDPR’s database is also not set up in accordance with best practices, as many entities have repeating information, making it difficult for the database team to maintain. Additionally, some relationships between entities in the database are not robust, scalable, or accurate to the actual relationship between entities in practice. Also, some attributes of entities are of inappropriate data types.

Furthermore, the legacy visitation application’s user interface is outdated, cluttered, and difficult to navigate. This creates a suboptimal experience for the park staff in their daily operations. NCDPR is in the process of updating its legacy applications and seeks to automate visitation data being sent from the IoT devices to the visitation application and improve user experience in the process.

## **Proposed Solution & Project Goals/Benefits**

Considering that the code base of the legacy visitation application is unorganized, unscalable, and written in an out-of-date version of PHP, our approach is to remake the visitation application instead of updating the existing code base. We will recreate the visitation application in PHP8 using the Slim framework, which will make the application significantly easier for NCDPR developers to maintain in the long term. Additionally, we will work with our sponsors and the other NC Parks and Recreation Senior Design team to create a unified backend code base for our finalized visitation application and the other updated applications of NCDPR’s LAMP stack, supporting a modernized stack in the process of migrating the existing LAMP stack. To address the structural problems of the database, we will redesign the database tables that are applicable to our project and work with the other NC Parks and Recreation Senior Design team to redesign the common tables that both of our applications will use. NCDPR will be able to start the process of migrating their data into the new unified database, as they are updating their legacy applications, at the conclusion of this project.

In upgrading the application, we will also automate the visitation data being transferred from the counters to the visitation application hourly, so that there is no need for park staff to manually input the data into the visitation application for the IoT counters that have the capabilities for automatic data transfer. However, our upgraded visitation application will still support functionality for park staff to manually input or edit visitation data for both counters without IoT capabilities or in scenarios where IoT counters are not accurately recording visitation data. We will also design a more user-friendly and modern interface, including various comparison metrics and data visualizations of park visitation.

# **Resources Needed**

*Authors: Shruti Jadhav, Quentin Sieredzki*

*Reviewers/Editors: Allison Church, Shruti Jadhav*

| **Resource Name** | **Purpose for Resource** | **Status** | **Version** | **Licensing Information** |
| --- | --- | --- | --- | --- |
| DPR Legacy Application | The existing legacy application will be used to better understand the functionality that we wish to recreate in our Visitation application, as well as ensure that everything is working correctly. | Obtained | N/A | Proprietary |
| Calendar Application | The project from last semester (with adjustments from Dr. Dominguez) will be used to understand the structure that our project should follow, as well as how to fit the two projects together. | Obtained | N/A | Proprietary |
| DB Schema / Data | The database will be redesigned to better access the relevant fields for our project, but the existing legacy database will also be necessary to understand what data is needed to include. | Obtained | N/A | Proprietary |
| Particle devices | The devices used to generate the visitation data. | Obtained | N/A | Provided through See Insights |
| Particle account | The account tied to the Particle devices used for additional resources potentially necessary. | Obtained | N/A | Provided through See Insights |
| Ubidots account | IoT platform that compiles the data from the Particle devices in a more digestible manner. | Obtained | N/A | Provided through See Insights |
| See Insights account | IoT vendor that uses Particle devices and the Ubidots platform in order to set up and deliver the Visitation information. The account will be used to get the data from the Ubidots platform. | Obtained | N/A | Provided through NCDPR |
| Docker engine | Containerization software used to help containerize our application (similar to Docker compose) in order to run multiple containers that interact with each other. | Obtained | 20.10.17 | Docker Personal License |
| Docker compose | Tool used to containerize our application within the existing environment of the DPR system, similar to last semester’s calendar application. | Obtained | 2.10.2 | Docker Personal License |
| Slim Framework | PHP micro-framework used to build our backend REST API. | Obtained | 4 | MIT |
| React | Frontend framework used in both our new Visitation application and the former team’s Calendar application. | Obtained | 17.0.2 | MIT |
| NGINX | Reverse Proxy used to route users to our new Visitation application. | Obtained | 1.20.2 | BSD |
| PHP | Scripting language used for the backend of our application. | Obtained | 8 | Open Source PHP License |
| PHPUnit | Unit testing framework used to test our PHP backend. | Obtained | 9 | BSD |
| Apache | Runtime environment used for the containers. | Obtained | 2.4.52 | Apache License |
| MariaDB | Database used for our application. | Obtained | 10 | GPLv2 |

*Table 2: List of Necessary Resources*

# **Risks & Risk Mitigation**

*Author: Allison Church*

*Reviewer/Editor: Shruti Jadhav, Mathew Deel*

The following list describes aspects of the project that may pose a risk to our team being able to complete the project over the course of the semester, as well as our plan to mitigate these risks:

* **Database Redesign and Migration:** Our team, the Senior Design technical advisors, and the NCDPR developers all agree that a database redesign is necessary for the long-term success of NCDPR’s migrated stack. NCDPR has decided to start the database redesign and migration process concurrently with the process of upgrading the LAMP stack, starting with our team’s redesigned sections of the database that the visitation application uses. We are also working with the other NC Parks and Recreation Senior Design team to redesign the common tables of the database that both our projects use, such as the tables for parks, applications, and users. We will not redesign sections of the database that other legacy applications use, as that is not within the scope of our project. However, it is guaranteed that our redesigned database tables will cause the functionality of legacy applications to break. To account for this and to properly integrate our redesigned sections of the database while still allowing other legacy applications to use older sections of the database, we will create a virtual table mechanism by using a view table in a SQL script. The view table will not store any data itself but will correspond to the segmented legacy and migrated database tables. We will thoroughly communicate our plan for the database with our sponsors and Senior Design technical advisors to minimize the risk of a poor transition from legacy to migrated database.
* **Poor Network Connectivity at Remote Parks:** The IoT counters rely on the cellular network to transfer their data to the Ubidots server. Sometimes devices are unable to upload their data at the scheduled time because of poor cellular network connectivity, which occurs frequently at the remote parks in particular. While devices are supposed to transfer their visitation data to the Ubidots server hourly, it may take multiple hours before a device has the connection to transfer its visitation data. When a device finally can transfer its data, it would ideally send all of this data together in a batch however this may not always be the case. After talking to the NC DPR team as well as SeeInsights, we discovered that problems could arise from this due to vaguely defined behavior that may cause some devices to act differently than others. In order to combat this, our system would be able to take this incoming data from the device with poor connection and parse it into multiple ‘Visit’ objects based on the last successful data transfer. From there, our functionality is being implemented in a flexible manner that allows our backend to easily accept an arbitrary number of visitation data logs regardless of the volume of data.

# **Development Methodology**

*Author: Shruti Jadhav*

*Reviewer/Editor: Quentin Sieredzki*

We used feature-driven development to carry out our implementation. Feature-driven development allows us to implement our solution in small pieces so that we can show progress to the sponsors and teaching staff in an iterative timeline. This is the most efficient methodology for us, as our project deals with many features such as adding/editing devices, adding/editing visitation data, pulling the data, and displaying the data. Due to some project redesigns, we kept the exact iteration length flexible with a rough plan of three weeks for each iteration. The actual iteration lengths and dates will also be covered in our implementation section and in each iteration we are focusing on implementing a feature by continuously revisiting the requirements, design, implementation, and testing. Our sponsor meetings are being held weekly on Mondays. This gives us the full week and weekend to make progress and then share our progress at the beginning of the next week. We also have weekly team meetings on Tuesdays and Thursdays where we give updates on our individual progress to the rest of our teammates.

# **System Requirements**

*Author: Mathew Deel*

*Reviewers/Editors: Alex Bowen, Allison Church*

**Overall View**

The Visitation Application will be an extension of the current legacy system and will be used exclusively by park staff internally as a tool for managing park visitation data. Users will be able to navigate to the Visitation App from the NC Parks and Recreation homepage in accordance with the other apps that they use. From there, users must authenticate themselves by logging in to verify that they are park staff members with valid credentials. Our team has created a set of temporary users with the no access, base level, manager, app admin, and super admin permission levels which work with our temporary authentication solution. In production our authentication will be reworked by NC DPR and they will use users with a shared set of permissions across all of the various supported apps. Once logged in, users are shown an interactive dashboard of visitation statistics for all parks overall which can be filtered down to a single park. On this dashboard, users will be able to navigate to a separate page that contains information about the IoT devices used to collect this data. Authenticated users will be able to add, edit, and remove certain IoT devices based on their level of permissions as well as counter rules which will automatically multiply data collected by devices based on its timestamp. Additionally, all users are able to manually add raw data to a device to account for non-IoT compatible devices or edit/delete post-calculated data to account for possible erroneous or accidental data entry. The calculation visit data The dashboard of statistics provided by this application will give users the ability to analyze this data to help conduct future planning and funding allocation.

## **Glossary**

## User - Any employee of NCDPR. Each user has a specific access level from the following list. Note that these are listed in increasing order of privilege:

## Level 0 - No Access (Denied access to the application upon login)

## Level 1 - Base

## Level 2 - Manager

## Level 3 - Admin

## Level 4 - Super-Admin

## IoT Device: Any given device, in this case measuring visitation numbers, which can connect over the internet to other devices to exchange/upload data.

## Dashboard: The data viewing page which describes visitation stats from various years with comparison metrics between them.

* Counter Rule: A property of a device that will take the raw count of any visit it collects which falls within its date range and then multiply it by a decimal number in order to accommodate for things like vehicles which carry more than 1 person.

## **Functional Requirements**

* **FR1 Login**
  + **FR1.1**. The system shall allow all users to log in using a username and password.
  + **FR1.2.** The system shall require all users to use an existing account from the legacy system with existing app permission levels.
  + **FR1.3.** The system shall restrict Manager or higher users to add/edit/delete devices.
  + **FR1.4.** The system shall restrict Admin or higher users to add/edit/delete counter rules.
* **FR2 Logout**
  + **FR2.1.** The system shall allow all users to log out at any time
* **FR3 View Device List**
  + **FR3.1.** The system shall allow all users to view a list of all registered devices.
  + **FR3.2.** The system shall distinguish all devices by their park code, device name, function, type, method, brand, multiplier, and date updated
  + **FR3.3.** The system shall allow all users to filter the list of devices based on a specific park.
* **FR4 Add IoT Device**
  + **FR4.1.** The system shall allow Manager users to add a new IoT device to any park.
  + **FR4.2.** The system shall require the entrant to enter the device’s name, number, SeeInsights ID, multiplier, park code, function, type, method, model, brand, latitude, and longitude
  + **FR4.3.** The system shall require the entrant to re-enter device information if the form has invalid/incomplete information.
* **FR5 Edit IoT Device**
  + **FR5.1.** The system shall allow Manager users to edit any visitation device on the device list.
  + **FR5.2.** The system shall require the entrant to input updated information or leave existing information for any required fields.
  + **FR5.3.** The system shall require the entrant to re-enter device information if the form has invalid/incomplete information.
* **FR6 Delete IoT Device**
  + **FR6.1.** The system shall allow Manager users to delete any visitation device on the device list.
* **FR7 View Counter Rule List**
  + **FR7.1.** The system shall allow all users to view a list of all counter rules associated with a device.
  + **FR7.2.** The system shall allow all users to view the time range for when the counter rule applies and the multiplier value associated with that rule.
* **FR8 Add Counter Rule**
  + **FR8.1.** The system shall allow Admin users to add a rule for any device.
  + **FR8.2.** The system shall require that each counter rule has an associated time range and multiplier.
* **FR9 Edit Counter Rule**
  + **FR9.1.** The system shall allow Admin users to edit any counter rule associated with a device.
  + **FR9.2.** The system shall require the entrant to input updated time range and multiplier information or leave it the same.
  + **FR9.3.** The system shall require the entrant to re-enter counter rule information if the form has invalid/incomplete information.
* **FR10 Delete Counter Rule**
  + **FR10.1.** The system shall allow Admin users to delete any counter rule for any device.
* **FR11 Add Raw Visitation Data**
  + **FR11.1.** The system shall allow Administrator users to add raw visitation data manually for any device.
  + **FR11.2.** The system shall allow Park Staff users to add raw visitation data manually for any device from their assigned park.
  + **FR11.2.** The system shall require the entrant to input a day timestamp, total count, and associated device for each visit log.
  + **FR11.3.** The system shall require the entrant to re-enter visit log information if the form has invalid or incomplete information.
  + **FR11.4.** The system shall reset the form after submission to allow for the entry of additional visit logs.
* **FR12 View Calculated Visitation Data**
  + **FR12.1.** The system shall allow all users to view the calculates visit logs after counter rules have been applied of any visitation device on the device list.
  + **FR12.2.** The system shall display the list of logs as their associated timestamps and counts.
* **FR13 Edit Calculated Visitation Data**
  + **FR13.1.** The system shall allow all users to edit any calculated visit.
  + **FR13.2.** The system shall allow the entrant to provide a visit with an updated visitation count.
  + **FR13.3.** The system shall require the entrant to re-enter visit log information if the visitation count is invalid/incomplete.
  + **FR13.4.** The system shall not allow any users to edit a visit log’s timestamp.
* **FR14 Delete Calculated Visitation Data**
  + **FR14.1.** The system shall allow Administrator users to delete any log on a device’s visitation data list.
  + **FR14.2.** The system shall allow Park Staff users to delete any visit log on a device’s visitation data list from their assigned park.
* **FR15 View Park Dashboard**
  + **FR15.1.** The system shall allow all users to view a dashboard containing visitation information for a park over time.
  + **FR15.2.** The system shall require that all dashboard data comes from the calculated visitation data which is derived from default multipliers and counter rules.
  + **FR15.3.** The system shall provide Park Staff users with the dashboard for their associated park by default.
  + **FR15.4.** The system shall provide Administrator users with the combined park dashboard by default.
  + **FR15.5.** The system shall allow all users to change which park is being displayed on the dashboard.
  + **FR15.6.** The system shall allow all users to change what timeframe is being displayed on the dashboard.
* **FR16 View Comparison Metrics**
  + **FR16.1.** The system shall allow all users to compare a park’s calculated visitation counts between two points in time.
  + **FR16.2.** The system shall allow all users to choose a timeframe over which data should be compared (day/month/year).
  + **FR16.3.** The system shall allow all users to choose two unique timestamps to compare.
* **FR17 Pull Data from Ubidots**
  + **FR17.1.** The system shall store all current data and archived historical data stored on Ubidots.
  + **FR17.2.** The system shall use the list of IoT-compatible devices to automatically pull data from Ubidots whenever new data is created.
  + **FR17.3.** The system shall use the Ubidots data to update its existing database

## **Non-Functional Requirements**

* **NFR1.** The visitation app should be integrated with the legacy system to be navigable from the landing page.

## **Constraints**

* **C1.** The Visitation App shall be containerized using Docker alongside the legacy applications to allow for different versions of PHP.
* **C2.** The Visitation App Backend shall be written in PHP8 using the Slim Framework.
* **C3.** The Visitation App Frontend shall be written in React using the MaterialUI library.
* **C4.** The Visitation App shall access the data from the Particle devices via the Ubidots cloud database.
  + **C4.1.** The Visitation App must not exceed the API call limit for the Ubidots token.
* **C5.** The Visitation App must be built off of the existing database using MariaDB 10 using the InnoDB engine.

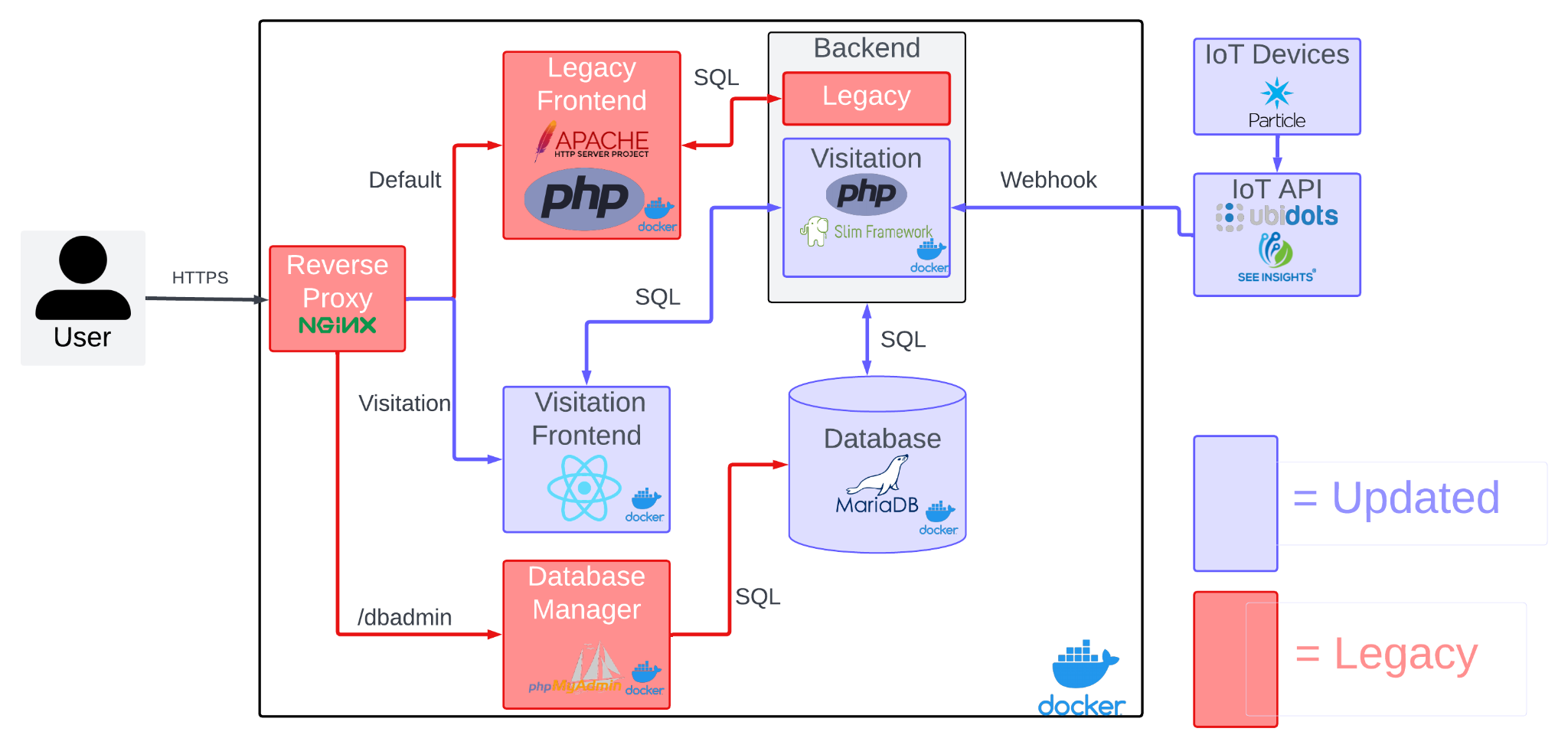
# **Design**

*Authors: Alex Bowen, Allison Church, Mathew Deel, Shruti Jadhav, Quentin Sieredzki*

*Reviewers/Editors: Mathew Deel, Allison Church*

## **High-Level Design**

The general architecture of the program is that the Department of Parks and Recreation Visitation (DPR Visitation) application will continue on the work of the previous Department of Parks and Recreation Calendar (DPR Cal) application in converting the existing DPR Legacy stack into becoming more modern. The major system components are the Reverse Proxy, the DPR Legacy application, the Database, the DPR Visitation Frontend, and the unified Backend shared by all DPR applications. The users will interact with the system through the Reverse Proxy, which then will redirect users into their respective application. If the user is using the DPR Visitation application, they will interact with the DPR Visitation Frontend. This will in turn be connected with the DPR Visitation Backend, which will handle all of the data calls with the Database and the IoT devices. Data will mainly be inputted from the IoT devices, but certain users with admin permission will be able to directly input information into the Database through the Database manager.



*Figure 1: DPR Visitation High-Level Architecture*

Docker Compose

The DPR Visitation architecture will continue to build upon the previous DPR Cal project by continuing to make new containers inside of Docker Compose. Figure 1 demonstrates the basic structure that will be used by the DPR Visitation system, and how it will fit into the ecosystem designed by the previous team. This allows us to continue their work of making the system more modular and less of a monolith design, facilitating an agile development process. Inside of the stack, different containers will be used for the DPR Legacy application, the DPR Visitation frontend, the DPR Visitation backend, the database, and more.

Reverse Proxy (Application Entrypoint)

Continuing on the previous group’s project, the entry point into the entire DPR application stack will continue to be an Nginx reverse proxy. This accepts HTTPS traffic and redirects it into the proper container for the application. Specifically, it will be used to serve the legacy DPR stack, the new DPR Visitation frontend, the new DPR Visitation backend, the previous group’s DPR Cal frontend, and the previous group’s DPR Cal backend on the same host. In the future, more services will continue to be allowed to be redirected through this proxy.

Visitation Frontend

The frontend for the new Visitation application will be a React app that runs in its own container. Information from the backend will be called from the backend container via its REST API, similar to the previous team’s implementation. This will allow for the frontend to be quicker, and only request and display necessary information.

Backend (REST API)

The backend of the new Visitation application will be provided by a REST API built on the SLIM PHP framework. The backend for all of the DPR applications will be hosted inside of one container, in order to make it easier for different applications to use the same code. This API will allow for the frontend to request HTTP requests to access data for the application, most notably the data incoming from the IoT devices. The IoT devices use the Ubidots platform, and the backend will interact with this through API calls to obtain the necessary data (see the IoT Devices section below). This data will be processed by the backend, and then imputed into (or extracted from) the database as necessary in order to store information for future reference.

IoT Devices

The DPR Visitation application will use IoT devices in order to obtain the information necessary. The IoT devices used are Particle devices, and the data that these collect will be given to the Ubidots platform for us to access. Our team will then use the Ubidots platform in order to retrieve the information that the devices have collected.

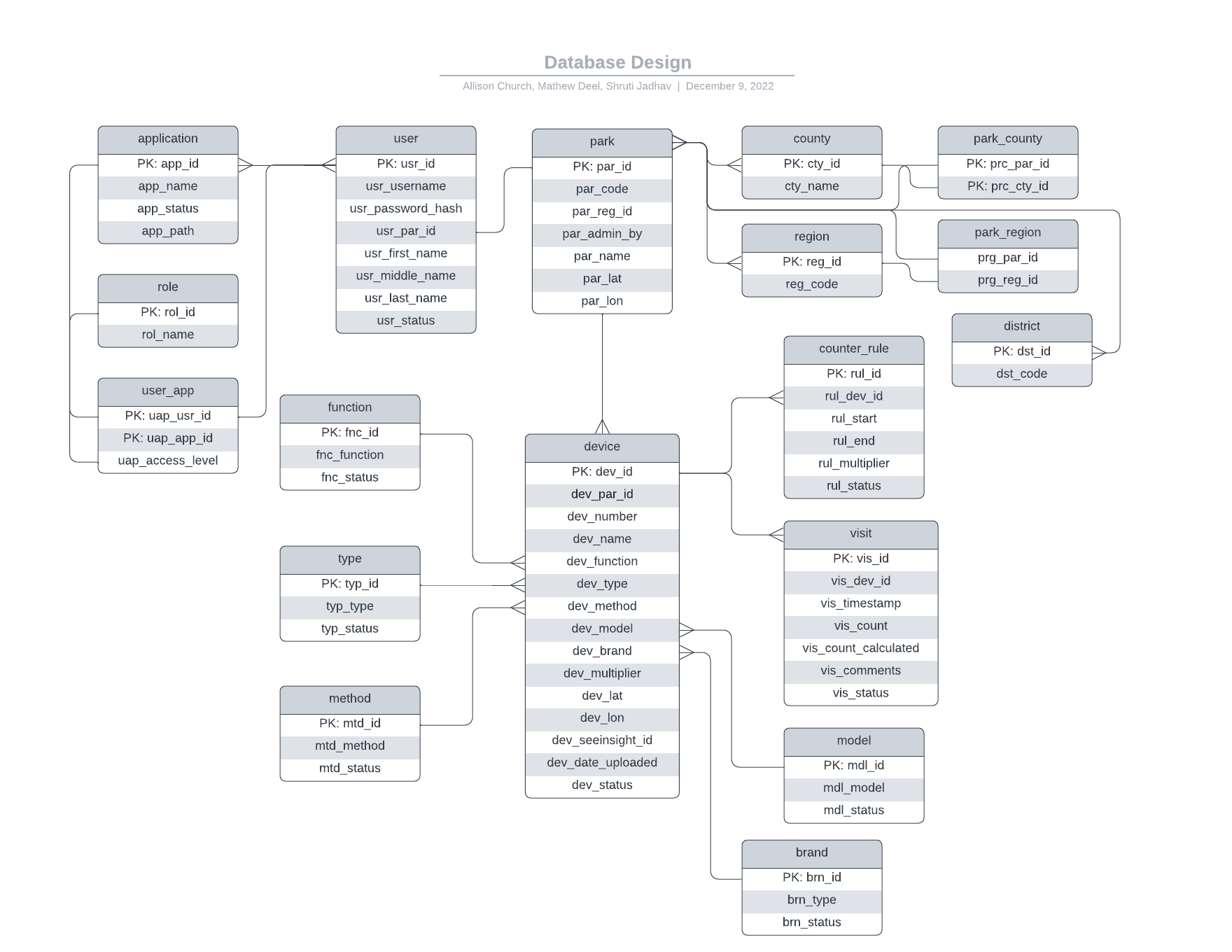
MariaDB Database

MariaDB will be used as the database for the system as its own container, continuing the setup that the previous group created. This allows for more modulation for different applications. Applications will have access to this through the DNS name, and the database administration will be handled by phpMyAdmin as necessary for the system administrators.

## **Low-Level Design**

Database Design

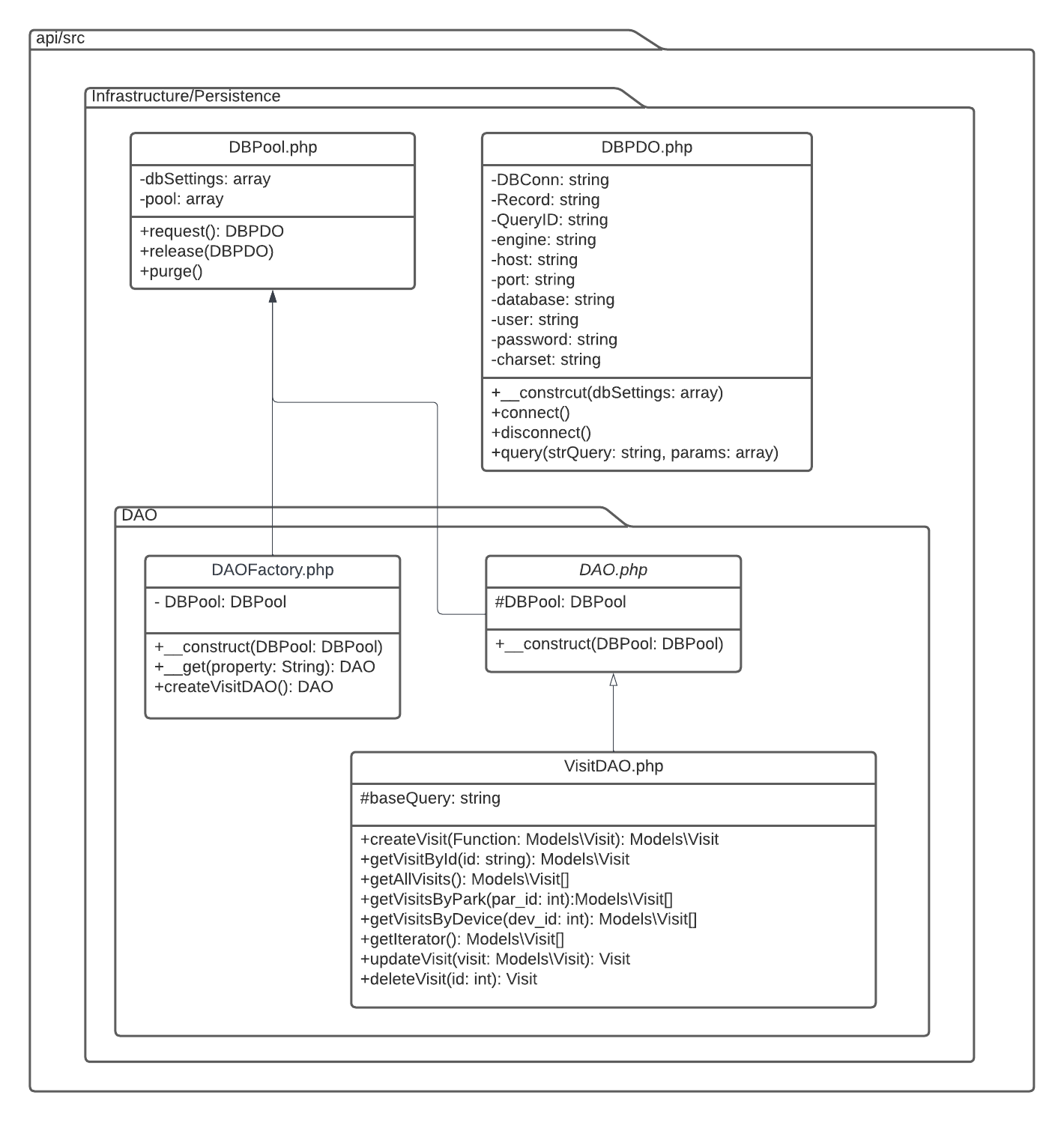
Figure 2 shows the database design for our Visitation application. When the **users** log in, they are associated with a single **park**, even if they are an admin who may work on several parks in real life. Many users can be associated with each park. Each park has multiple **devices** within it tracking the visitation data. The **type** table keeps track of all the device types and models and has many devices. The **function** table has many devices and keeps track of the functionality of the device, such as traffic or other functions. The **method** table keeps track of whether the device is gathering data manually or automatically. The **model** table has all of the possible device models. The **brand** table keeps track of the device brand. The **counter rule** table has multiple devices and the range rule is the time range for when this rule is applied and the multiplier which can be used to calculate the actual count of visitation. Each device has multiple instances of **visit** which tracks the timestamp that data was pulled, the original count of the device, and the calculated count using the multiplier.

**

*Figure 2: Visitation Database Design*

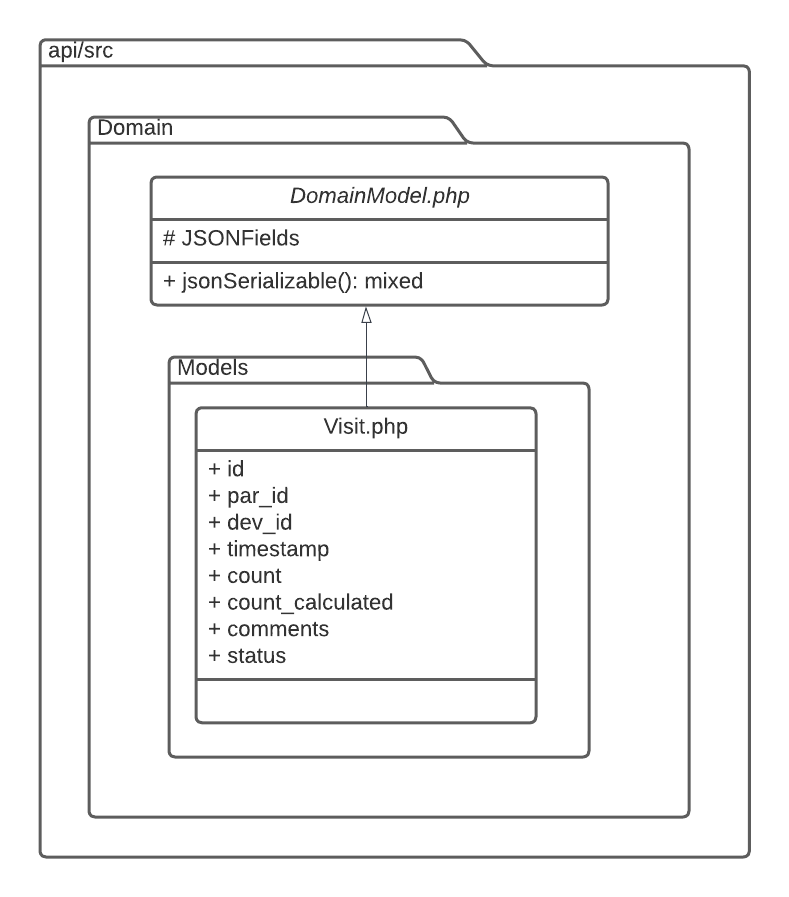
Unified Backend Design

The unified backend is contained in the api/src namespace and is further structured into application, domain, and infrastructure namespaces. The infrastructure namespace is a data access layer. It contains a DBPool class which maintains database connections and allows for parallel requests to the database. DBPool uses the DBDPO class to create the connections to and query the database directly. The infrastructure namespace also contains a DAO namespace for data access objects, which separate the data resource’s client interface from the data access mechanisms. While Figure 3 only shows the VisitDAO class, the following DAOs are also included, but are omitted from the diagram for brevity: AuthenticationDAO, BrandDAO, CounterRuleDAO, DeviceDAO, FunctionDAO, MethodDAO, ModelDAO, ParkDAO, and TypeDAO. All of these classes extend from the abstract *DAO* class and have functions to support creating, getting, updating, and deleting database entities based on the functionality specified by our API endpoints. DAOFactory is used to construct and get specific DAO classes.



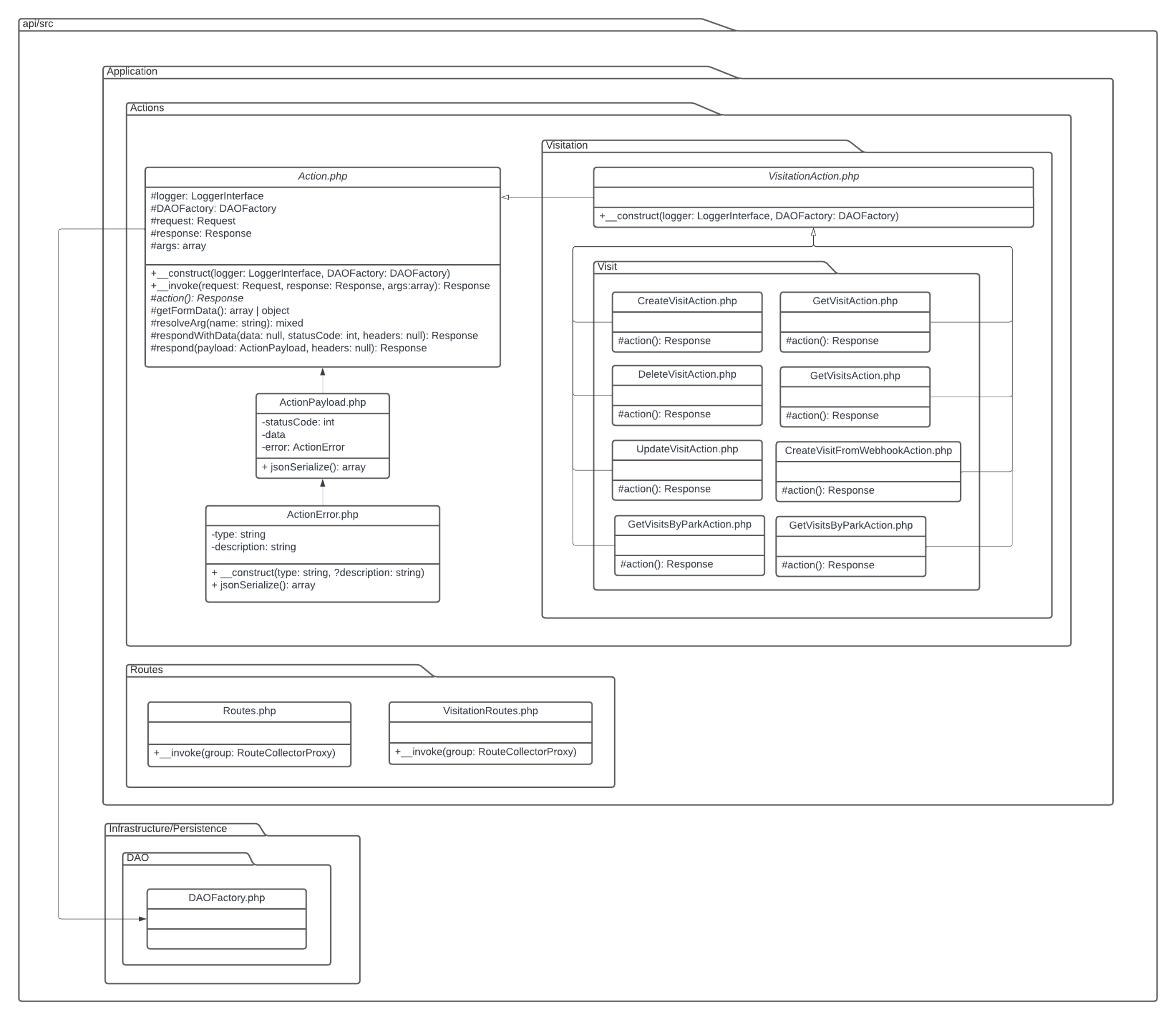
*Figure 3: UML Class Diagram for Data Access Layer*

The domain namespace also contains models of the database entities. Figure 4 shows the Visit class model for the Visit database entity with fields for the attributes. There are also model classes for the Brand, CounterRule, Device, Function, Method, Model, Park, and Type database entities, but they were excluded from the diagram for brevity. All of these model classes extend from an abstract *DomainModel* class that is JSON serializable.



*Figure 4: UML Class Diagram for Domain Layer*

The Application namespace contains a namespace for actions that support our API endpoint functionality. The abstract *Action* class contains an instance of DAOFactory, which it uses to handle the action in the data access layer when a request is made. *VisitationAction* is another abstract class that extends from *Action*, which contains an instance of UbidotsAPI, and acts as a structure for actions specific to visitation functionality. Within the Action namespace, there are namespaces for specific functionality relating to our API endpoints, including namespaces for Visit, Devices, Brands, CounterRules, Functions, Methods, Models, and Types. All actions within these namespaces extend from *VisitationAction*. Park actions are contained separately from the Visitation-specific actions, as Park actions are more general to the unified backend. Only Visit actions are shown in Figure 3 for brevity, but the remaining actions follow the same structure as Visit actions. There are Visit actions based on the API endpoint functionality of creating, getting, deleting, and updating a visit. The application namespace also contains the route classes that call actions as methods when they receive a request from the frontend.

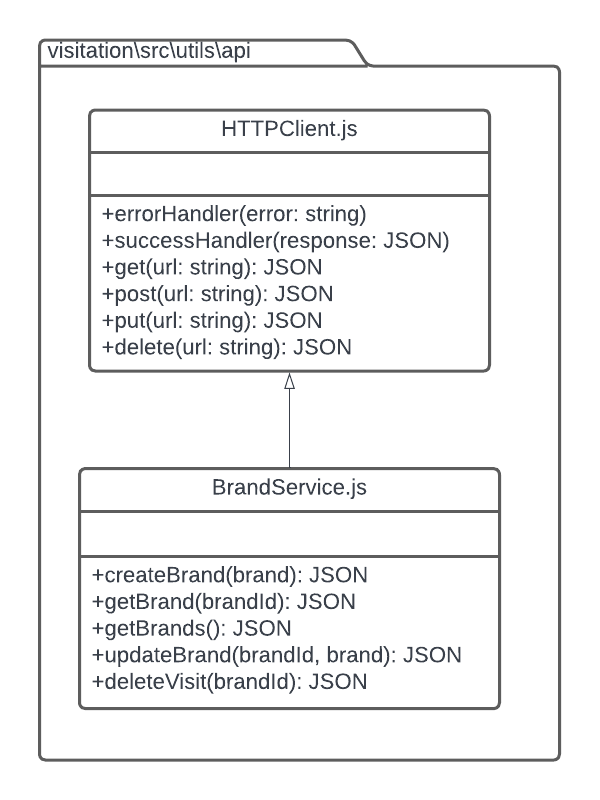


*Figure 5: UML Class Diagram for Application Layer*

Frontend Design

Our frontend was designed to have all our frontend pages in the Components folder with our Service classes in the utils. The service classes allow our Javascript pages to access the API routes needed to send data back and forth. We separated our components into 4 sections; Auth, Dashboard, Devices, and Data. Each of these sections contains the associated pages needed to make up that component. In each page we are importing React and MUI components and have functions to handle changes in the data. Most pages utilize the useEffect() function to get the updated data. We also have the page returning all the HTML components needed. When moving data between pages we are using the navigate function to pass on the state of any objects that are needed. Our frontend also uses session storage to keep track of user roles and permissions so that specific pages or components are displayed based on user level. The interactive dashboard was made using the Apex Charts library and uses a JSX file so that multiple components can be seen on the same page.

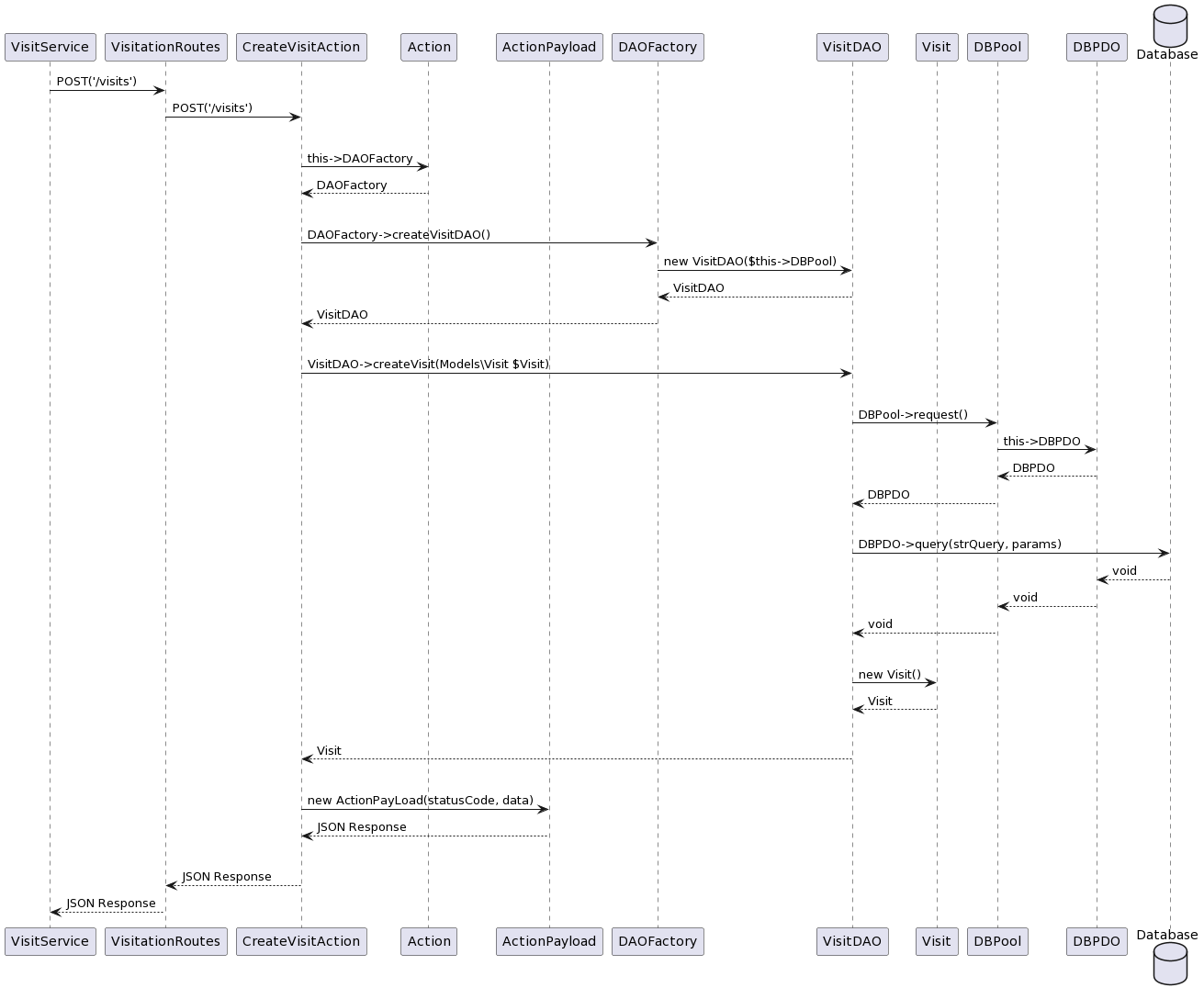
The frontend sends all requests to the backend through service Javascript classes to organize HTTP requests and promises. The services classes extend from an HTTPClient Javascript class, which details CRUD operations for the requests and has an error handler as well. There are service classes specifically for authentication, brands, counter rules, devices, functions, methods, models, parks, types, and visits. These classes contain functions that align with and call specific routes of our application and return a JSON from the routes to the frontend ReactJS components. Figure 6 below shows the BrandService.js class specifically, but the rest of the service classes follow the same design.



*Figure 6: UML Class Diagram for Frontend Service Classes*

UML Sequence Diagram

Figure 7 below shows the flow of the creation of a Visit. Our application’s frontend uses VisitService to redirect requests to the VisitationRoutes class, which then calls the appropriate *Action* subclass, CreateVisitAction, as a method. CreateVisitAction receives the DAOFactory from its parent class, *Action*. CreateVisitAction then uses the DAOFactory to create a new VisitDAO. CreateVisitAction then uses that VisitDAO to call the create function. To do this, VisitDAO first uses DBPool to request a new database connection. DBPool returns a DBPDO instance for the database connection. VisitDAO then uses that DBPDO instance to query the database to create the visit. If a visit does not already exist, the database connection returns void, and the query was successful. VisitDAO can then construct a new Visit model and return that new Visit model to the CreateVisitAction class. The CreateVisitAction class then creates a new ActionPayload with a successful status code and data about the newly created Visit and returns a JSON response, which persists to the VisitationRoutes and then to the VisitService classes.

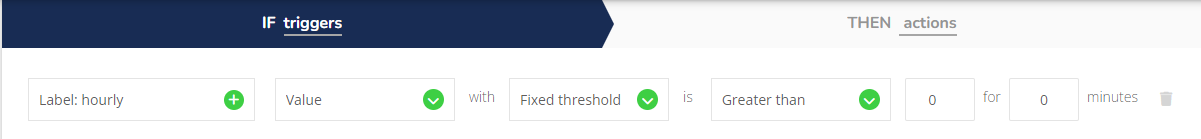


*Figure 7: UML Sequence Diagram for Visit Creation*

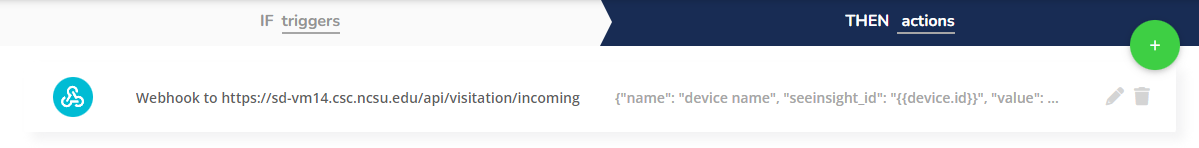
While only one flow of an API call is shown in the diagram above, all API calls for our application follow a very similar flow, and are excluded for brevity. Our application’s frontend components use a specific service class to handle requests. The service classes redirect to the appropriate route which calls a specific *Action* subclass. That subclass uses its parent *Action* class’ instance of DAOFactory to create a corresponding *DAO* subclass. The appropriate CRUD method of that *DAO* subclass is called, and the *DAO* subclass will use DBPool to obtain a DBPDO connection to query the database. A corresponding Model class may be used to represent an instance of an entity and returned to the *Action* subclass that was called. The *Action* subclass will use the ActionPayload class to respond with an appropriate status code and data. The JSON response will propagate back up to the service class.

Ubidots Webhook

Part of our design involves the webhook which exists on the Ubidots cloud server. All devices have been grouped together based on the park that they are associated with in order to make them more convenient to work with. For each group, we create one webhook hosted on the SeeInsights web server which consists clause which is formatted as an “IF triggers THEN actions” clause as shown below in Figures 8 and 9. The IF statement is designed so that the webhook is triggered whenever the hourly count for any device in the group is greater than 0. When this is true, the THEN statement will make the proper POST call to our routes using the data that triggered the webhook. This call will go through to the database on the NCSU virtual machine which is being hosted to host the Visitation application. It is important to note that we have specific routes for our API endpoints relating to Ubidots different than the other endpoints for general Visitation application functionality to be able to parse and handle the data from each source properly in different action classes.



*Figure 8: Webhook IF statement for Lake James State Park*

**

*Figure 9: Webhook THEN statement for Lake James State Park*

Rest API Endpoints

As shown in the Visitation Backend section, the REST API serves HTTP requests from the frontend that connect with the backend operations and the database. The functionality of the endpoints is centered on the ability of Users to add, and edit the attendance for parks as well as add, edit, and remove Devices that automatically populate the attendance.

There is an example group provided to demonstrate our paths. We have a group specific for authentication. The Parks group allows the user to retrieve all parks, or one park based on the park\_code. The Devices group will allow the user to send requests to add, edit, and delete the devices based on the device id and also retrieve all the devices in a park or all parks based on the park\_code in the query. The Visits group allows the user to retrieve the attendance for a certain date, month, year, hour or all time. The visits can also be retrieved for a specific park, multiple parks, or all parks and can be added or edited when the user inputs a specific park code and exact date [2]. The CounterRules group allows the user to retrieve counter rules based on the deviceid. The rules can be updated, created, deleted, and retrieved. The CounterRules group allows the user to retrieve counter rules based on the deviceid. The rules can be updated, created, deleted, and retrieved.

The CounterRules group allows the user to retrieve counter rules based on the deviceid. The rules can be updated, created, deleted, and retrieved. Type, Function, Brand, Model, and Method are all part of a device. They are given individual groups as well so that multiple devices can be updated at a time. These groups are able to be retrieved by id.

**Example Group:**

**/path: GET, POST**

**/{path\_variable}: GET, PUT, DELETE**

**/path: GET, POST**

**/path/{path\_variable}: GET**

**Unified NCDPR Base Path:**

**/api**

**Authentication:**

**/auth**

Login endpoint for the Calendar application.

**/token: POST**

Login endpoint for the Visitation application.

- Note: A different login endpoint was created for the Visitation application to point to a different action class that follows the updated design for the unified backend (using DAOFactory to connect to the database) that Calendar applications’ login action class does not follow.

**/login: POST**

-Response: {Response: “”, User: “”, User Role: “”, User Parks: [{}], result: “”}

**Users:**

Note: These Users endpoints are used for the Calendar application, but are not used by the Visitation application.

Get all users, post all users

**/users: GET, POST**

Get a user, update a user, and delete a user, using their id

**/{id}: GET, PUT, DELETE**

**Parks:**

Get all parks

**/parks:GET**

-Response: {parks: [ARCH:{}, BRKA:{}, UMSD:{}]}

Get a park by id

**/parks/{par\_id}:GET**

-Response: {park: ARCH:{}}

Get park by park code

**/parks/code{par\_code}:GET**

-Response: {park: ARCH:{}}

**Visitation Application Base Path:**

**/api/visitation**

**Devices:**

Get all devices and create a device

**/devices: GET, POST**

-Response: {[ex\_device\_1:{...}, ex\_device\_2:{...}, ex\_device\_3:{...}]}

Get a device, update a device, delete a device by id

**/devices/{dev\_id}: GET, PUT, DELETE**

-Response: { [ex\_device\_1:{...}, ex\_device\_2:{...}, ex\_device\_3:{...}]}

Get all devices in a specific park

**/parks/{par\_id}/devices: GET**

-Response: {ARCH: [ex\_device\_1:{...}, ex\_device\_2:{...}, ex\_device\_3:{...}]}

**Visits:**

Get all visits

**/visits: GET**

{[{vis\_1:{vis\_timestamp: “3:00:05”, vis\_count: 4, vis\_count\_calculated: 8}}, {vis\_2: {vis\_timestamp: “6:00:05”, vis\_count: 1, vis\_count\_calculated: 2}}]

Get all visits aggregated into months

**/visits/month: GET**

{[{vis\_1:{vis\_timestamp: “3:00:05”, vis\_count: 4, vis\_count\_calculated: 8}}, {vis\_2: {vis\_timestamp: “6:00:05”, vis\_count: 1, vis\_count\_calculated: 2}}]

Get visit by visit id

**/visits/{vis\_id}: GET**

{vis\_1:{vis\_timestamp: “3:00:05”, vis\_count: 4, vis\_count\_calculated: 8}}

Get visits in a specific device and create visit for specific device

**/devices/{dev\_id}/visits: GET, POST**

-Response: {exdevicename: [{vis\_1:{vis\_timestamp: “3:00:05”, vis\_count: 4, vis\_count\_calculated: 8}}, {vis\_2: {vis\_timestamp: “6:00:05”, vis\_count: 1, vis\_count\_calculated: 2}}]}

Update and delete visit in a specific device

**/devices/{dev\_id}/visits/{vis\_id}: PUT, DELETE**

Get all visits in a specific park

**/parks/{par\_id}/visits: GET**

-Response: {ARCH: [ex\_device\_name:[{vis\_1:{vis\_timestamp: “3:00:05”, vis\_count: 4, vis\_count\_calculated: 8}}, {vis\_2: {vis\_timestamp: “6:00:05”, vis\_count: 1, vis\_count\_calculated: 2}}, ex\_device\_name2:[{vis\_1:{vis\_timestamp: “5:00:05”, vis\_count: 4, vis\_count\_calculated: 8}}, {vis\_2: {vis\_timestamp: “4:00:05”, vis\_count: 1, vis\_count\_calculated: 2}}]]}

Get all devices in a specific park aggregated into days

**/parks/{par\_id}/visits/day: GET**

-Response: {ARCH: [ex\_device\_name: {vis\_1:{vis\_timestamp: “3:00:05”, vis\_count: 4, vis\_count\_calculated: 8}}]}

**Counter Rules:**

Get all counter rules and create a counter rule

**/counter\_rules: GET, POST**

**-**Response**:** {[{counter\_rule\_1:{rul\_start: “January”, rul\_end: “May”, rul\_multiplier: 3}}, {counter\_rule\_2:{rul\_start: “June”, rul\_end: “December”, rul\_multiplier: 2}}]}

Get all counter rules from a specific device

**/devices/{device\_id}/counter\_rules: GET**

**-**Response**:** {ex\_device\_name: [{counter\_rule\_1:{rul\_start: “January”, rul\_end: “May”, rul\_multiplier: 3}}, {counter\_rule\_2:{rul\_start: “June”, rul\_end: “December”, rul\_multiplier: 2}}]}

Get a counter rule, update a counter rule, and delete a counter using counter rule id

**/counter\_rules/{counter\_rule\_id}: GET, PUT, DELETE**

**-**Response**:** {counter\_rule\_1:{rul\_start: “January”, rul\_end: “May”, rul\_multiplier: 3}}

**Types:**

Get all types and create a type

**/types: GET, POST**

-Response: {types: [{type: “infrared”}, {type: “pressure”}, {type: “bodyheat”}]}

Get a type, update a type, delete a type using a type id

**/type/{typ\_id}: GET, PUT, DELETE**

-Response: {ex\_device\_name: [{type: “infrared”}]}

**Function:**

Get all functions and create a function

**/functions: GET, POST**

-Response: {functions: [{function: “traffic”}, {function: “trails”}, {function: “visitor center”}]}

Get a function, update a function, and delete a function using function id

**/functions/{fun\_id}: GET, PUT, DELETE**

-Response: {ex\_device\_name: [{function: “traffic”}]}

**Brands:**

Get all brands and create a brand

**/brands: GET, POST**

-Response: {brands: [{brand: “Particle”}]}

Get a brand, update a brand, delete a brand using brand id

**/brands/{brn\_id}: GET, PUT, DELETE**

-Response: {ex\_device\_name: [{brand: “Particle”}]}

**Models:**

Get all models and create a model

**/models: GET, POST**

-Response: {models: [{model: “AV456”}, {model: “AV896”}, {model: “AV452”}]}

Get a model, update a model, delete a model using model id

**/models/{mdl\_id}: GET, PUT, DELETE**

-Response: {ex\_device\_name: {model: “AV456”}}

**Methods:**

Get all methods and create a method

**/methods: GET, POST**

-Response: {methods: [{method: “manual”}, {method: “automatic”}]}

Get a method, update a method, delete a method, using method id

**/methods/{mtd\_id}: GET, PUT, DELETE**

-Response: {ex\_device\_name: {method: “manual”}}

**Ubidots:**

Get all devices from the Ubidots server

**/fetch/devices: GET**

Get all visits from Ubidots server (long runtime, ~1-2 hours)

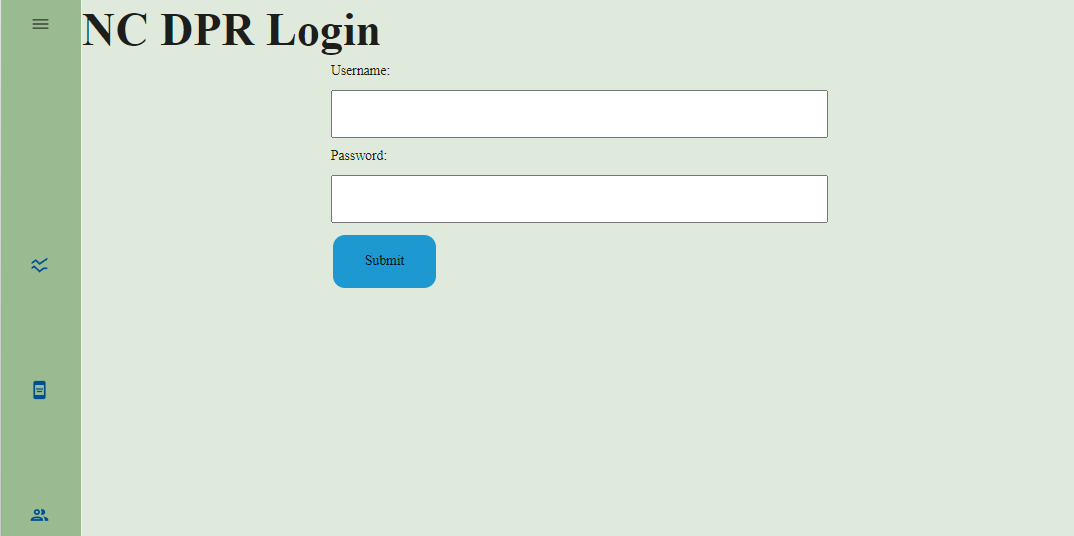
**/fetch/visits: GET**

Creates visits from data pulled from the webhook

**/incoming: POST**

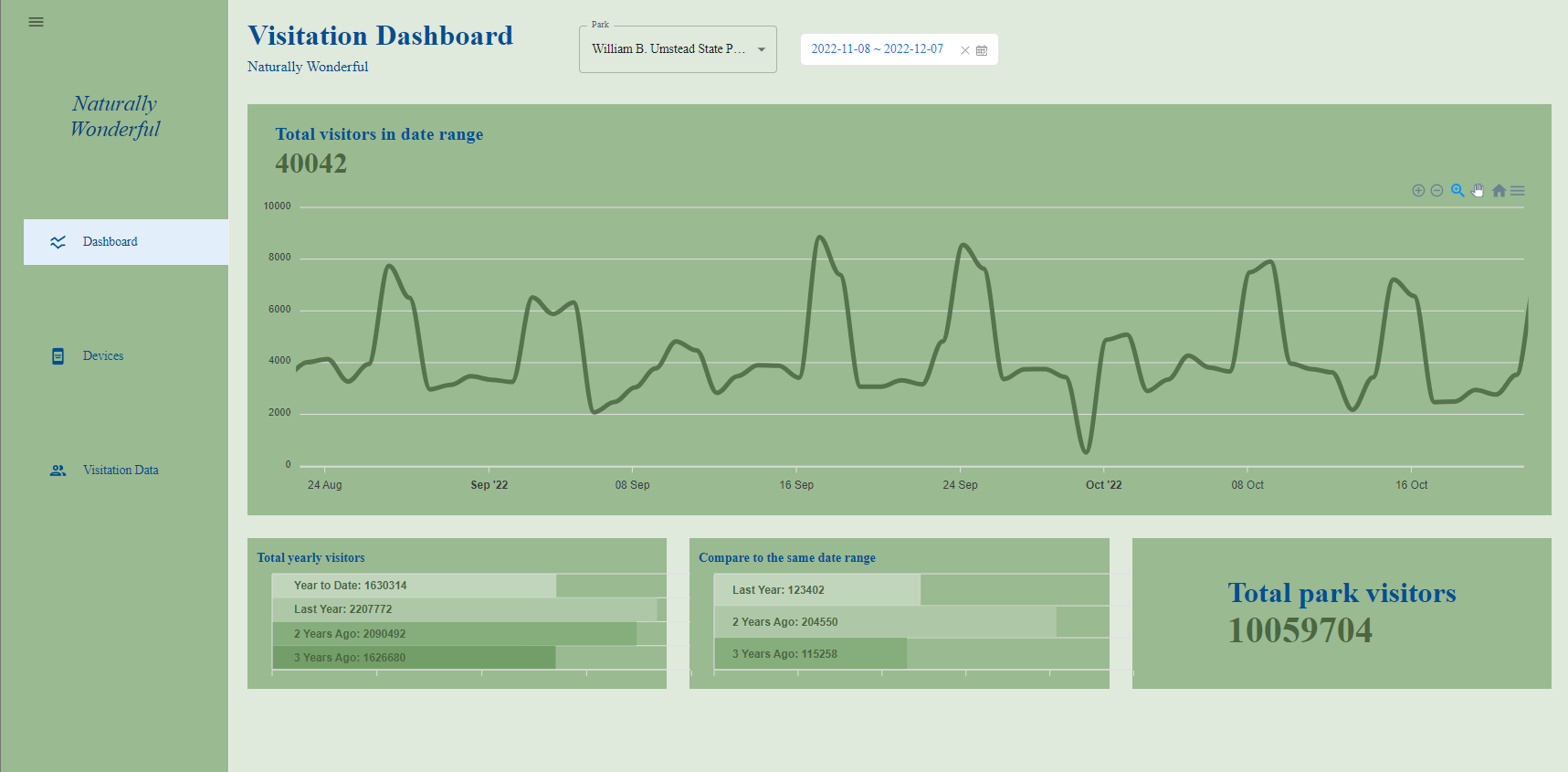
**GUI Design**

Upon navigating to the Visitation application via the NCDPR homepage, the user will be prompted to enter their username and password to verify their credentials before continuing on to the application. A message will be displayed if any incorrect login information is submitted. At any point, the user may click the “Dashboard” button to return to the NCDPR homepage.



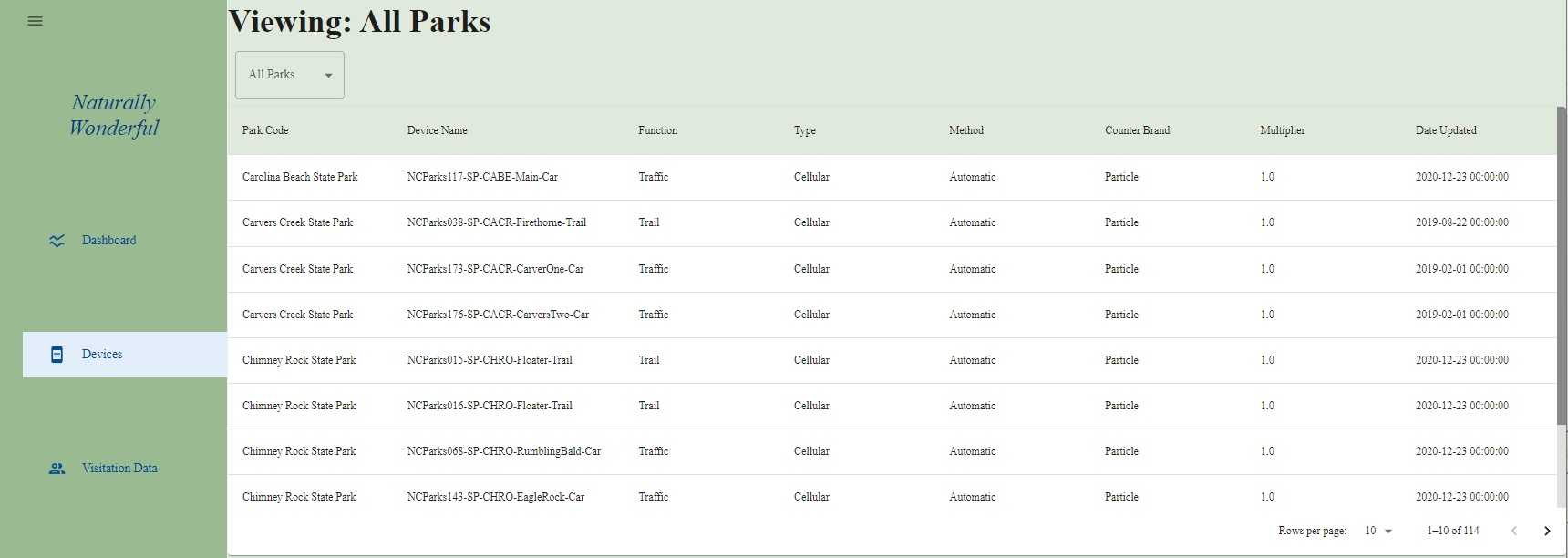
*Figure 10: Visitation Application Login Page*

The landing page of the application contains the visitation statistics dashboard. In contrast to the legacy visitation app, visitation numbers will be displayed in the form of a line graph that represents total visitor counts over one year. Upon entering the app for the first time, the user will be shown the cumulative visitation counts for their park, visually seeing the changes over the last month. The user will have the option to select a range of dates and a particular park (or “all parks”) when viewing the dashboard. At the bottom, the user also sees relevant stats about their park.



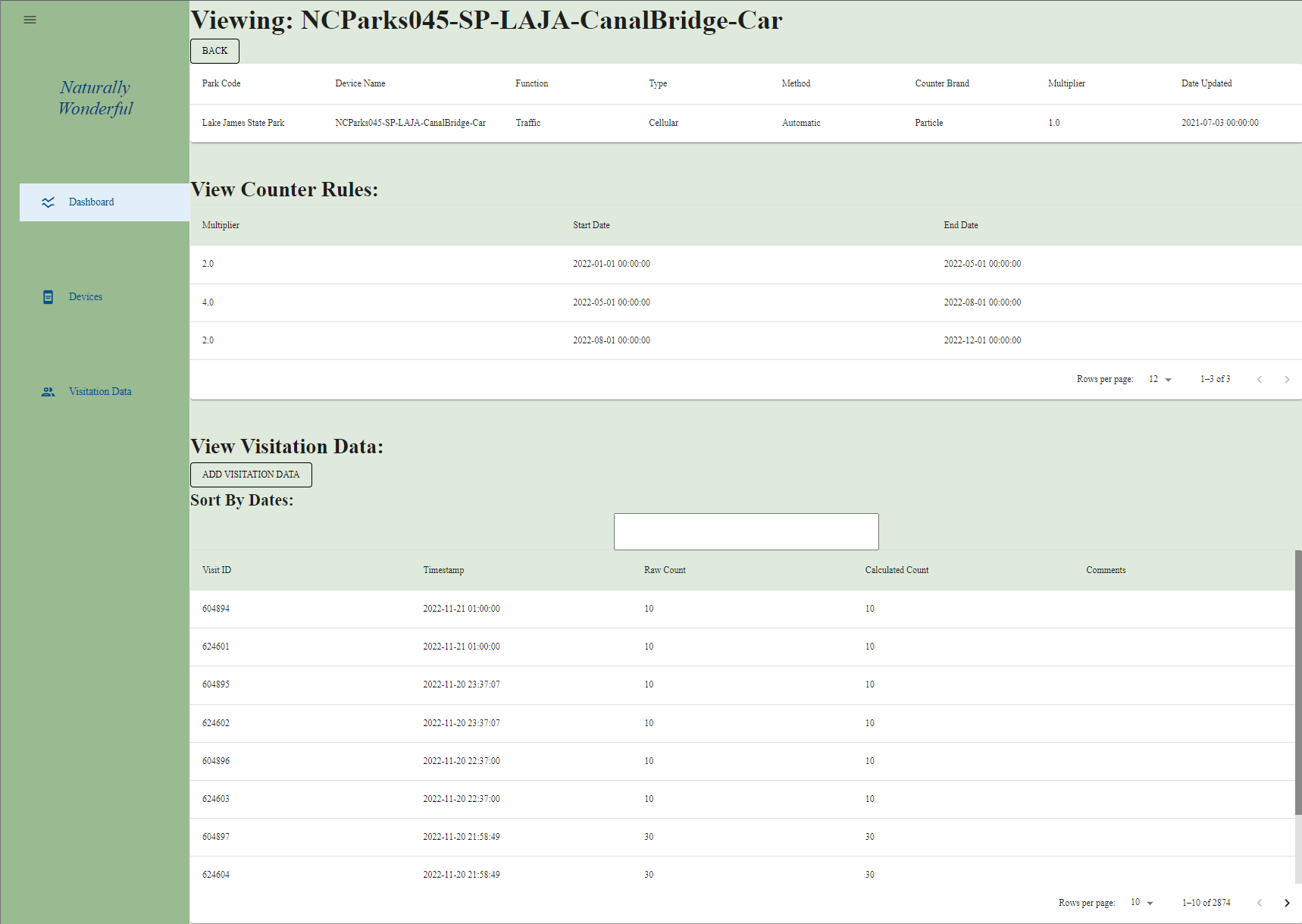
*Figure 11: Visitation Dashboard*

Users also have the options of selecting between ‘Devices’ and ‘Visitation Data’ by using the left dropdown menu. If the user selects on ‘Devices’, they will be able to access a list of all of the IoT devices for a park. If the user has the proper authentication level, they will be able add a new device. Clicking on a device will bring them to a page showing that device’s information (as shown in Figure 10.)



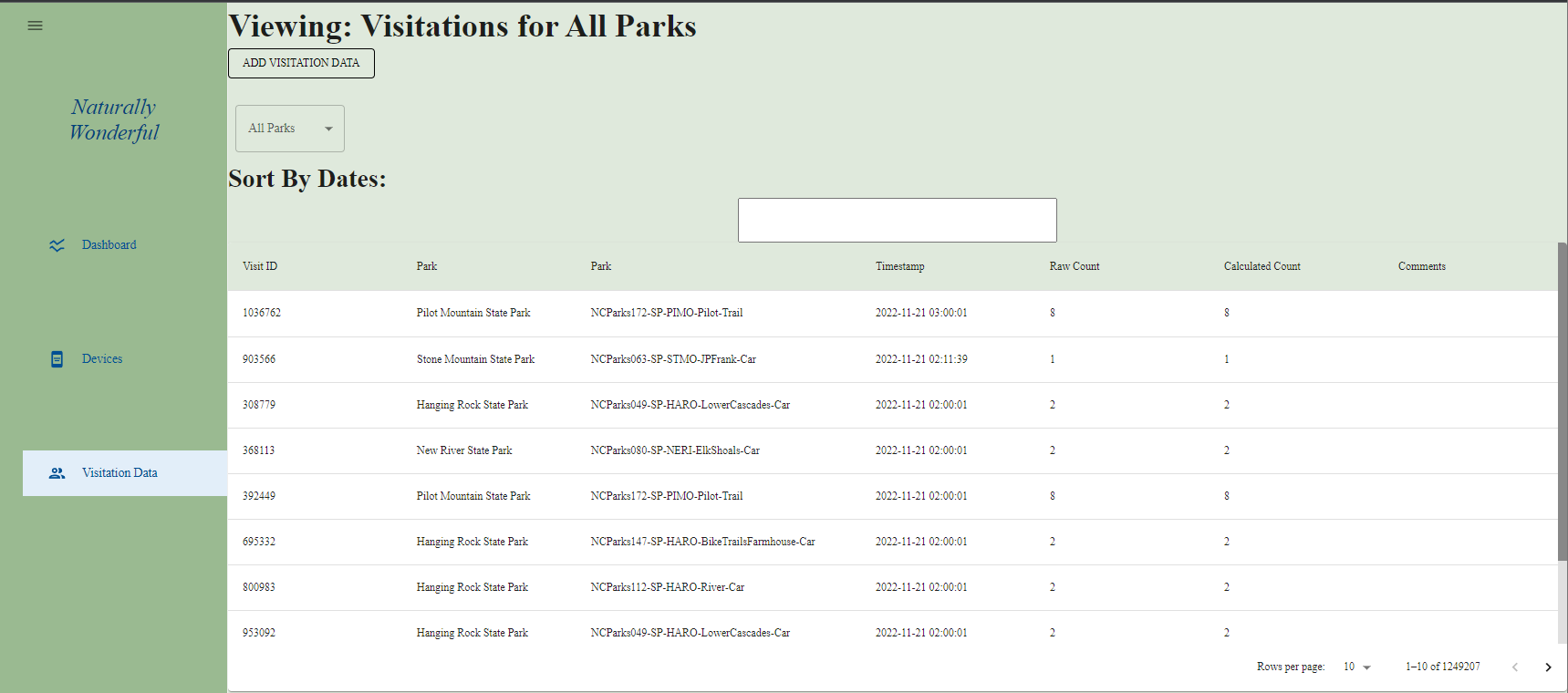
*Figure 12: View Devices Page*

After selecting a specific device, the user can see the details of the device. They can see the general information about the device, as well as the counter rules associated with the device. They can also see the Visitation Data associated with the device. From this screen, there is also the option to edit the device and counter rules (if the user has the proper permissions), as well as adding or editing Visitation Data.

****

*Figure 13: View Device Page*

Finally, from the Visitation Data page, the user can see all of the visits for their park (or all parks). From this page, they can sort by certain dates, edit the visitation data, and add new visitation data.



*Figure 14: View Data Page*

# **Implementation**

*Authors: Alex Bowen, Allison Church, Mathew Deel, Shruti Jadhav*

*Reviewers/Editors: Alex Bowen, Quentin Sieredzki*

## **Iteration Definition & Current Status**

* Iteration 0 (Work done before starting iterations)
  + Start date: 8-22-22
  + End date: 9-18-22
  + Features:
    - The focus of this iteration was on getting the existing application running and configured on each of our local machines
      * Compose the application in Docker - Completed
      * Make sure that dependencies are installed - Completed
      * Explore the legacy system and calendar app - Completed
* Iteration 1 (Current Iteration)
  + Start date: 9-19-22
  + End date: 10-7-22
  + Features:
    - The focus of this iteration is to work on migrating the old codebase and getting a front-to-back connection running with sample display data
    - Update directory structure according to Slim Framework and software development best practices
      * [All FRs] - Completed
    - Convert the shared User/Application tables to new database as the basis for user permission and authentication
      * [FR1] [FR2] - Completed
    - Create the central device table as well as the connected function, type, counter rule, visit, and calculated visit tables
      * [FR3] [FR7] [FR12] - Completed
    - Create a basic front-to-back connection between the Docker containers as a framework for moving visitation data between layers
      * [All FRs] - Completed
    - Pull and display sample data from the Ubidots API as a proof of concept
      * [FR17] - Completed
* Iteration 2
  + Start date: 10-10-22
  + End date: 10-30-22
  + Features:
    - The focus of this iteration is to work on getting all functionality pertaining to the device list working on the backend with basic frontend app navigation implemented in order to reach the relevant pages
    - Allow users to add/edit/delete devices in the system with types/functions
      * [FR3] [FR4] [FR5] [FR6] - Completed
    - Create basic frontend for app navigation with device pages
      * [FR3] [FR4] [FR5] [FR6] - Completed
    - Allow users to add raw uncalculated data to the system
      * [FR11] - Completed
    - Migrate in the old visitation data to database
      * [FR12] - Completed
* Iteration 3
  + Start date: 10-31-22
  + End date: 11-20-22
  + Features:
    - The focus of this iteration is to build off of the existing devices functionality any needed functionality such as counter rules to prepare the application to deal with actual visitation data
    - Implement counter rules with conditional multipliers based on timestamp
      * [FR7] [FR8] [FR9] [FR10] - Completed
    - Calculate new visitation information based on counter rules and raw counter data
      * [FR12] - Completed
    - Automatically pull in data from Particle devices as well as allow manual data entry
      * [FR17] - Completed
    - Start creating a basic dashboard to view the migrated data with
      * [FR15] - Completed
    - Add additional frontend pages in accordance with the complete wireframe
      * [All FRs] - Completed
* Iteration 4
  + Start date: 11-21-22
  + End date: 12-9-22
  + Features:
    - The focus of this iteration is to finalize out the existing functionality as well and focus on the data visualization aspect with the dashboard
    - Allow the users to manually edit or delete calculated visitation data
      * [FR13] [FR14] - Completed
    - Flesh out the existing dashboard and include comparison metrics with it
      * [FR15] [FR16] - Completed
    - Finalize the frontend design in order to maximize usability
      * [All FRs] - Completed
    - Fix any bugs discovered in acceptance testing
      * [All FRs] - Completed

## **Security Considerations**

* **Identification & Authentication**
  + **User Authentication:** While our sponsors have plans to overhaul the authentication for their LAMP stack, we still needed to create a basic authentication system for users. Users will authenticate into the system using a username and password. The Argon2 hashing algorithm is used with a salt to store passwords, making it resistant to GPU and side-channel attacks. Upon login, a JWT is generated with the user’s username, role, and parks included in the payload. Our JWT implementation is similar to the Calendar application’s login, which is signed using the same secret key stored in the project’s secrets folder. Like the Calendar application, the JWT is split into three separate cookies for the header, payload, and signature, and are stored in different cookies in the client’s browser via the Set-Cookie response header for security. While the header and payload are stored in regular cookies, the signature is stored using the HttpOnly attribute to protect against cross-site request forgery attacks. The cookies expire after a day.

All of our applications’ routes have a *TokenAuthMiddleware* class added to them, except those relating to the Ubidots webhook which use a separate middleware class for authentication. The *TokenAuthMiddlware* class screens the requests for the JWT cookies and redirects to the route only if the cookies are valid.

* + **Ubidots Webhook Authentication:** Our application relies on publicly exposed routes to allow Ubidtos to send requests to transfer visitation data to our application. With publicly exposed routes come potential security vulnerabilities. To address security concerns, we coordinated with the President of SeeInsights, Chip McClelland, to include NCDPR’s read-only API key in the header of requests sent from the Ubidots webhook. Our application’s three HTTP routes relating to Ubidots API endpoint functionality have a *UbidotsAuthMiddlware* class added that automatically screens incoming HTTP requests to validate that the request is from Ubidots. If the key is not present in the header of the request or the key does not match our Ubidots API key that is stored as an environment variable upon project startup, then an *HTTPUnauthorizedException* is thrown. *UbidotsAuthMiddleware* checks for a matching API key using the PHP hash\_equals function to protect the API key and our API endpoint from timing attacks [3]. If the API key provided in the header is correct, then the middleware proceeds to route the request to the proper action class to handle the desired functionality.

Additionally, our sponsors ensured us that NCDPR also has a firewall on their deployed LAMP stack applications, which blocks network traffic other than that from Ubidots. It is also noteworthy that the webhook security concerns are not regarding the visitation data, as the data being sent from Ubidots is considered “public information” of the state parks system, but more so just preventing a potential attacker from falsifying information by calling our application’s endpoints.

* **Availability of Resources**
  + **User-Role Permissions:** Essentially all data regarding the Visitation application is considered “public information” of NCDPR (information surrounding parks, devices, and visitation), so the application does not necessarily deal with any “sensitive” data. Therefore, our application does not restrict authenticated users from viewing any of this data. However, our application does restrict some users’ ability to add, edit, or delete devices and counter rules, as specified in our application’s requirements. This is configured by placing permissions on our application’s routes and screening for the appropriate permissions in the *TokenAuthMiddleware* class. If an authenticated user does not have the appropriate user-level access role (that is stored in the JWT payload cookie), then the *TokenAuthMiddlware* class will throw an *HTTPForbiddenException* with an unauthorized status code.

## **Project Folder Structure**

To continue the modulation of the NCDPR suite of applications, we have decided to have a separate folder for our frontend ReactJS files, but have a unified backend folder that contains many elements from the Slim Framework basic skeleton structure, including folders for action classes, domain logic, and middleware [4]. A basic overview of the different project folders that we will be using is described below.

2022/FallTeam06-NC-Parks-2

* api

# The unified backend of migrated applications.

* + app

# Contains configuration files of the backend,

# including settings, dependencies, repositories, etc.

* + public

# Contains the index file, provided by the Slim

# skeleton, to provide scaffolding for the backend.

* + src

# Source code for the unified backend.

* + - Application
      * Actions

# Contains action classes for application

# functionality.

* + - * + Auth

# Actions specific for authentication.

* + - * + Parks

# Actions specific for park

# functionality.

* + - * + Visitation

# Actions specific for the Visitation

# functionality. Contains subfolders

# for actions based more specific

# Visitation API functionality.

Brands

CounterRules

Devices

Functions

Methods

Models

Types

Visit

* + - * Routes

# Contains files to route API calls to

# specific actions.

* + - * Middleware

# HTTP middleware added to routes to filter # requests to check authentication. Includes # middleware classes for Ubidots specific

# routes and all other generic routes.

* + - * Handlers
      * ResponseEmitter
      * Settings

# The Handlers, ResponseEmitter, and

# Settings folders are for project

# scaffolding.

* + - Domain

# Business logic layer of the application.

* + - * Domain Exception

# Contains classes for domain exceptions.

* + - * Models

# Contains model classes to reflect database

# entities.

* + - Infrastructure

# The database access layer of the backend.

* + - * Persistence

# Contains the classes that facilitate

# connections to the database.

* + - * + DAO

# Folder for data access objects,

# including specific classes to

# correspond to all of the model

# entities.

* + tests

# Folder for backend unit testing.

* db

# Contains database and database management files.

* + data

# Contains updated and legacy database files.

* + db\_schema

# Contains database schema descriptor files to

# populate the database with empty schema objects

# when needed.

* legacy\_application

# Contains the legacy applications for reference.

* nginx

# Contains configuration settings for the Nginx web server.

* secrets

# Contains sensitive project information.

* visitation

# Contains frontend files for the Visitation Application,

# including the Dockerfile to build the frontend visitation # container.

* + public

# Contains public files that the frontend uses, such

# as logos.

* + src

# Contains the source files for the Visitation # Application frontend.

* + - components

# Contains ReactJS functional component files for # the Visitation Application.

* + - contexts

# Contains the context files needed to

# bootstrap the Visitation application.

* + - utils
      * api

# Contains JS service files to route to

# specific API calls to the unified backend.

## **Project Configuration/Settings**

The following are specific components of our project configuration setup:

* **Environment File:** The environment file (dev.env) sets up global constants that will be accessible to and be used by applications during runtime like system hostnames, configurations, or credentials. Having these settings stored in an environment file allows for project settings to be changed simply. Different environment files can be created for different purposes, for example, development or deployment. Our environment file is specifically for development.

Our environment file specifically holds database configurations and credentials, Xdebug configurations for unit testing coverage, and the Ubidots URL and API key to validate requests from Ubidots. The following shows the outline of what is stored in our environment file.

DEFAULT\_HOSTNAME=

DEPLOY\_TYPE=

MYSQL\_ROOT\_PASSWORD=

MYSQL\_USER=

MYSQL\_PASSWORD=

DB\_ENGINE=

DB\_HOST=

DB\_PORT=

DB\_USER=

DB\_PASSWORD=

DB\_CHARSET=

PUBLIC\_URL=

#XDEBUG

XDEBUG\_MODES=

XDEBUG\_LOG\_LEVEL=

DPR\_API\_BASE=

#UBIDOTS

UBIDOTS\_DATASOURCES\_URL=

UBIDOTS\_API\_KEY=

* **Secrets Folder:** The secrets folder contains sensitive configuration settings. This includes the private key and signed SSL certificate that is used to enable encrypted connections for the NCDPR system, as well as the JSON Web Token secret key. These components are excluded from our GitHub repository by a .gitignore file. The secrets must be configured by the user for deployment.
* **Docker**
  + docker-compose.yml: This file details the multi-container Docker setup that is used to host NCDPR’s LAMP stack, including containers for the reverse-proxy web server Nginx, the administrative database management tool phpMyAdmin, the MariaDB database, the legacy applications, the Calendar application frontend, the Visitation application frontend, and the unified backend. Having separate Docker containers for all of these elements to version these elements in isolation.
  + Dockerfile: The dockerfile contains instructions for assembling images with the correct application source code and environment setup. This is used on the calendar frontend, visitation frontend, and legcay applications’ containers.

# **Testing**

*Author: Mathew Deel*

*Reviewers/Editors: Quentin Sieredzki, Alex Bowen*

## **Overall View**

The Visitation Application project will use a combination of both unit testing and acceptance testing in order to verify its functionality. The unit testing will test our backend components by leveraging PHPUnit in order to run a series of automated white box tests however, this could be further expanded in the future to have frontend unit testing for React. This will assist in making sure that once our API endpoints are queried, our backend is actually parsing visitation data properly and updating the MariaDB database accordingly.

Conversely, our acceptance testing will be conducted manually via a test plan of black box tests. The goal for acceptance tests is to verify that all of our outlined functional requirements exist in the current state of the application and are fully usable as described. We will not need to configure a separate dev environment to run the testing environment dataset specified below. The only existing, persistent data that needs to be in the system before running the tests is just the various user accounts. The rest of the data will be created while running the test cases.

| Testing Environment Dataset | |
| --- | --- |
| Base Level user | * Username: permission-1 * Password: perm-1!sdc06NCPARKS2 * Current Park: WIUM |
| Super Admin user | * Username: permission-4 * Password: perm-4!sdc06NCPARKS2 * Current Park: WIUM |

## In order to set up the user accounts, you must navigate to localhost/dbadmin and go to divper>emplist. From here you can find the permission-4 and permission-1 users and edit their park and password. These users represent different levels of permissions and the park associated with a user affects the output of the application in certain use cases. This process should only need to be done once because the database is persistent and will retain these changes.

## **Unit Testing**

For the Visitation Application, we are using PHPUnit in order to test our backend as mentioned before. We are expecting to reach a total of 100% method coverage with 80% statement and branch coverage as a goal. However, the actual percentage is hard to calculate due to our coverage tool not filtering out untestable units These tests will focus on the ‘Visitation’ section of the backend which acts as a means of transporting device data queried from the API endpoints to both the MariaDB database as well as the end user.

There are some software components that will be excluded from our unit testing. Firstly, we will not unit test our GUIs because it relies too heavily on design decisions and will be tested in acceptance testing manually. Given extra time however, we would expand our unit testing to include a React testing library like JEST in order to validate that our functionality is working from front to back..

We also will not be testing the third-party Ubidots platform that we are pulling the Particle device data from. While their API endpoints are available to use, we are assuming that the JSON data output is correct and aligns with the expected behavior. Our testing instead focuses on the software pipeline that happens after the JSON data is collected.

# **Unit Testing Information**

Unit Testing Tool: PHPUnit

Coverage Tool: PHPUnit

Exempt Units: None Currently

# **Test Coverage Report**

## **OVERALL COVERAGE SUMMARY**

| **name** | **class, %** | **method, %** | **line, %** |
| --- | --- | --- | --- |
| All classes | 46.60%  (48/103) | 55.96%  (169/302) | 27.82%  (660/2372) |

### **OVERALL STATS SUMMARY**

| total packages: | 1 |
| --- | --- |
| total executable files: | 103 |
| total classes: | 103 |
| total methods: | 305 |
| total executable lines: | 2536 |

### **COVERAGE BREAKDOWN BY PACKAGE**

DPR\API\Application\Actions\Action

Methods: 50.00% ( 3/ 6) Lines: 75.00% ( 18/ 24)

DPR\API\Application\Actions\ActionError

Methods: 66.67% ( 4/ 6) Lines: 83.33% ( 10/ 12)

DPR\API\Application\Actions\ActionPayload

Methods: 60.00% ( 3/ 5) Lines: 84.62% ( 11/ 13)

DPR\API\Application\Actions\Auth\GetAuthenticationTokenAction

Methods: 0.00% ( 0/ 4) Lines: 0.00% ( 0/ 53)

DPR\API\Application\Actions\Files\Categories\GetCategoriesAction

Methods: 0.00% ( 0/ 1) Lines: 0.00% ( 0/ 4)

DPR\API\Application\Actions\Parks\GetDeviceIdBySeeInsightsIdAction

Methods: 0.00% ( 0/ 1) Lines: 0.00% ( 0/ 6)

DPR\API\Application\Actions\Parks\GetParkAction

Methods: 0.00% ( 0/ 1) Lines: 0.00% ( 0/ 6)

DPR\API\Application\Actions\Parks\GetParkIdByCodeAction

Methods: 0.00% ( 0/ 1) Lines: 0.00% ( 0/ 6)

DPR\API\Application\Actions\Parks\GetParksAction

Methods: 0.00% ( 0/ 1) Lines: 0.00% ( 0/ 5)

DPR\API\Application\Actions\User\ListUsersAction

Methods: 0.00% ( 0/ 1) Lines: 0.00% ( 0/ 3)

DPR\API\Application\Actions\User\UserAction

Methods: 0.00% ( 0/ 1) Lines: 0.00% ( 0/ 2)

DPR\API\Application\Actions\User\ViewUserAction

Methods: 0.00% ( 0/ 1) Lines: 0.00% ( 0/ 4)

DPR\API\Application\Actions\Visitation\Brands\CreateBrandAction

Methods: 100.00% ( 1/ 1) Lines: 100.00% ( 8/ 8)

DPR\API\Application\Actions\Visitation\Brands\DeleteBrandAction

Methods: 100.00% ( 1/ 1) Lines: 100.00% ( 6/ 6)

DPR\API\Application\Actions\Visitation\Brands\GetBrandAction

Methods: 100.00% ( 1/ 1) Lines: 100.00% ( 6/ 6)

DPR\API\Application\Actions\Visitation\Brands\GetBrandsAction

Methods: 100.00% ( 1/ 1) Lines: 100.00% ( 5/ 5)

DPR\API\Application\Actions\Visitation\Brands\UpdateBrandAction

Methods: 100.00% ( 1/ 1) Lines: 100.00% ( 9/ 9)

DPR\API\Application\Actions\Visitation\CounterRules\CreateCounterRuleAction

Methods: 0.00% ( 0/ 1) Lines: 0.00% ( 0/ 11)

DPR\API\Application\Actions\Visitation\CounterRules\DeleteCounterRuleAction

Methods: 0.00% ( 0/ 1) Lines: 0.00% ( 0/ 6)

DPR\API\Application\Actions\Visitation\CounterRules\GetCounterRuleAction

Methods: 0.00% ( 0/ 1) Lines: 0.00% ( 0/ 6)

DPR\API\Application\Actions\Visitation\CounterRules\GetCounterRulesAction

Methods: 0.00% ( 0/ 1) Lines: 0.00% ( 0/ 5)

DPR\API\Application\Actions\Visitation\CounterRules\GetCounterRulesByDeviceAction

Methods: 0.00% ( 0/ 1) Lines: 0.00% ( 0/ 6)

DPR\API\Application\Actions\Visitation\CounterRules\UpdateCounterRuleAction

Methods: 0.00% ( 0/ 1) Lines: 0.00% ( 0/ 12)

DPR\API\Application\Actions\Visitation\Devices\CreateDeviceAction

Methods: 0.00% ( 0/ 1) Lines: 0.00% ( 0/ 20)

DPR\API\Application\Actions\Visitation\Devices\DeleteDeviceAction

Methods: 0.00% ( 0/ 1) Lines: 0.00% ( 0/ 6)

DPR\API\Application\Actions\Visitation\Devices\FetchLegacyDevicesAction

Methods: 0.00% ( 0/ 1) Lines: 0.00% ( 0/ 10)

DPR\API\Application\Actions\Visitation\Devices\FetchLegacyVisitsAction

Methods: 0.00% ( 0/ 1) Lines: 0.00% ( 0/ 15)

DPR\API\Application\Actions\Visitation\Devices\GetDeviceAction

Methods: 0.00% ( 0/ 1) Lines: 0.00% ( 0/ 6)

DPR\API\Application\Actions\Visitation\Devices\GetDevicesAction

Methods: 0.00% ( 0/ 1) Lines: 0.00% ( 0/ 5)

DPR\API\Application\Actions\Visitation\Devices\GetDevicesByParkAction

Methods: 0.00% ( 0/ 1) Lines: 0.00% ( 0/ 6)

DPR\API\Application\Actions\Visitation\Devices\UpdateDeviceAction

Methods: 0.00% ( 0/ 1) Lines: 0.00% ( 0/ 21)

DPR\API\Application\Actions\Visitation\Functions\CreateFunctionAction

Methods: 100.00% ( 1/ 1) Lines: 100.00% ( 8/ 8)

DPR\API\Application\Actions\Visitation\Functions\DeleteFunctionAction

Methods: 0.00% ( 0/ 1) Lines: 0.00% ( 0/ 6)

DPR\API\Application\Actions\Visitation\Functions\GetFunctionAction

Methods: 100.00% ( 1/ 1) Lines: 100.00% ( 6/ 6)

DPR\API\Application\Actions\Visitation\Functions\GetFunctionsAction

Methods: 100.00% ( 1/ 1) Lines: 100.00% ( 5/ 5)

DPR\API\Application\Actions\Visitation\Functions\UpdateFunctionAction

Methods: 100.00% ( 1/ 1) Lines: 100.00% ( 9/ 9)

DPR\API\Application\Actions\Visitation\Methods\CreateMethodAction

Methods: 100.00% ( 1/ 1) Lines: 100.00% ( 8/ 8)

DPR\API\Application\Actions\Visitation\Methods\DeleteMethodAction

Methods: 100.00% ( 1/ 1) Lines: 100.00% ( 6/ 6)

DPR\API\Application\Actions\Visitation\Methods\GetMethodAction

Methods: 100.00% ( 1/ 1) Lines: 100.00% ( 6/ 6)

DPR\API\Application\Actions\Visitation\Methods\GetMethodsAction

Methods: 100.00% ( 1/ 1) Lines: 100.00% ( 5/ 5)

DPR\API\Application\Actions\Visitation\Methods\UpdateMethodAction

Methods: 100.00% ( 1/ 1) Lines: 100.00% ( 9/ 9)

DPR\API\Application\Actions\Visitation\Models\CreateModelAction

Methods: 100.00% ( 1/ 1) Lines: 100.00% ( 8/ 8)

DPR\API\Application\Actions\Visitation\Models\DeleteModelAction

Methods: 100.00% ( 1/ 1) Lines: 100.00% ( 6/ 6)

DPR\API\Application\Actions\Visitation\Models\GetModelAction

Methods: 100.00% ( 1/ 1) Lines: 100.00% ( 6/ 6)

DPR\API\Application\Actions\Visitation\Models\GetModelsAction

Methods: 100.00% ( 1/ 1) Lines: 100.00% ( 5/ 5)

DPR\API\Application\Actions\Visitation\Models\UpdateModelAction

Methods: 100.00% ( 1/ 1) Lines: 100.00% ( 9/ 9)

DPR\API\Application\Actions\Visitation\Types\CreateTypeAction

Methods: 100.00% ( 1/ 1) Lines: 100.00% ( 8/ 8)

DPR\API\Application\Actions\Visitation\Types\DeleteTypeAction

Methods: 100.00% ( 1/ 1) Lines: 100.00% ( 6/ 6)

DPR\API\Application\Actions\Visitation\Types\GetTypeAction

Methods: 100.00% ( 1/ 1) Lines: 100.00% ( 6/ 6)

DPR\API\Application\Actions\Visitation\Types\GetTypesAction

Methods: 100.00% ( 1/ 1) Lines: 100.00% ( 5/ 5)

DPR\API\Application\Actions\Visitation\Types\UpdateTypeAction

Methods: 100.00% ( 1/ 1) Lines: 100.00% ( 9/ 9)

DPR\API\Application\Actions\Visitation\VisitationAction

Methods: 0.00% ( 0/ 1) Lines: 0.00% ( 0/ 2)

DPR\API\Application\Actions\Visitation\Visits\CreateVisitAction

Methods: 0.00% ( 0/ 1) Lines: 0.00% ( 0/ 30)

DPR\API\Application\Actions\Visitation\Visits\CreateVisitFromWebhookAction

Methods: 0.00% ( 0/ 1) Lines: 0.00% ( 0/ 31)

DPR\API\Application\Actions\Visitation\Visits\DeleteVisitAction

Methods: 0.00% ( 0/ 1) Lines: 0.00% ( 0/ 6)

DPR\API\Application\Actions\Visitation\Visits\GetDayVisitsByParkAction

Methods: 0.00% ( 0/ 1) Lines: 0.00% ( 0/ 6)

DPR\API\Application\Actions\Visitation\Visits\GetMonthVisitsAction

Methods: 0.00% ( 0/ 1) Lines: 0.00% ( 0/ 5)

DPR\API\Application\Actions\Visitation\Visits\GetVisitAction

Methods: 100.00% ( 1/ 1) Lines: 100.00% ( 6/ 6)

DPR\API\Application\Actions\Visitation\Visits\GetVisitsAction

Methods: 100.00% ( 1/ 1) Lines: 100.00% ( 5/ 5)

DPR\API\Application\Actions\Visitation\Visits\GetVisitsByDeviceAction

Methods: 0.00% ( 0/ 1) Lines: 0.00% ( 0/ 6)

DPR\API\Application\Actions\Visitation\Visits\GetVisitsByParkAction

Methods: 0.00% ( 0/ 1) Lines: 0.00% ( 0/ 6)

DPR\API\Application\Actions\Visitation\Visits\UpdateVisitAction

Methods: 0.00% ( 0/ 1) Lines: 0.00% ( 0/ 13)

DPR\API\Application\Handlers\HttpErrorHandler

Methods: 0.00% ( 0/ 1) Lines: 61.54% ( 16/ 26)

DPR\API\Application\Handlers\ShutdownHandler

Methods: 0.00% ( 0/ 2) Lines: 0.00% ( 0/ 30)

DPR\API\Application\Middleware\SessionMiddleware

Methods: 0.00% ( 0/ 1) Lines: 50.00% ( 2/ 4)

DPR\API\Application\Middleware\TokenAuthMiddleware

Methods: 0.00% ( 0/ 1) Lines: 0.00% ( 0/ 20)

DPR\API\Application\ResponseEmitter\ResponseEmitter

Methods: 0.00% ( 0/ 1) Lines: 0.00% ( 0/ 12)

DPR\API\Application\Routes\AuthRoutes

Methods: 0.00% ( 0/ 1) Lines: 66.67% ( 2/ 3)

DPR\API\Application\Routes\CalendarRoutes

Methods: 0.00% ( 0/ 1) Lines: 71.26% ( 62/ 87)

DPR\API\Application\Routes\FilesRoutes

Methods: 100.00% ( 1/ 1) Lines: 100.00% ( 1/ 1)

DPR\API\Application\Routes\ParkRoutes

Methods: 100.00% ( 1/ 1) Lines: 100.00% ( 6/ 6)

DPR\API\Application\Routes\Routes

Methods: 100.00% ( 1/ 1) Lines: 100.00% ( 6/ 6)

DPR\API\Application\Routes\UserRoutes

Methods: 0.00% ( 0/ 1) Lines: 66.67% ( 16/ 24)

DPR\API\Application\Routes\VisitationRoutes

Methods: 100.00% ( 1/ 1) Lines: 100.00% (105/105)

DPR\API\Application\Settings\Settings

Methods: 100.00% ( 2/ 2) Lines: 100.00% ( 2/ 2)

DPR\API\Domain\DomainException\DomainException

Methods: ( 0/ 0) Lines: ( 0/ 0)

DPR\API\Domain\DomainException\DomainRecordNotFoundException

Methods: ( 0/ 0) Lines: ( 0/ 0)

DPR\API\Domain\DomainModel

Methods: 100.00% ( 1/ 1) Lines: 100.00% ( 5/ 5)

DPR\API\Domain\Models\Brand

Methods: 100.00% ( 7/ 7) Lines: 100.00% ( 16/ 16)

DPR\API\Domain\Models\CounterRule

Methods: 100.00% (13/13) Lines: 100.00% ( 28/ 28)

DPR\API\Domain\Models\Device

Methods: 100.00% (31/31) Lines: 100.00% ( 64/ 64)

DPR\API\Domain\Models\DeviceFunction

Methods: 100.00% ( 7/ 7) Lines: 100.00% ( 16/ 16)

DPR\API\Domain\Models\File

Methods: 0.00% ( 0/ 1) Lines: 0.00% ( 0/ 7)

DPR\API\Domain\Models\Method

Methods: 100.00% ( 7/ 7) Lines: 100.00% ( 16/ 16)

DPR\API\Domain\Models\Model

Methods: 100.00% ( 7/ 7) Lines: 100.00% ( 16/ 16)

DPR\API\Domain\Models\Park

Methods: 100.00% (15/15) Lines: 100.00% ( 32/ 32)

DPR\API\Domain\Models\Type

Methods: 100.00% ( 7/ 7) Lines: 100.00% ( 16/ 16)

DPR\API\Domain\Models\Visit

Methods: 100.00% (17/17) Lines: 100.00% ( 36/ 36)

DPR\API\Domain\Ubidots\UbidotsAPI

Methods: 0.00% ( 0/ 2) Lines: 0.00% ( 0/ 90)

DPR\API\Domain\Ubidots\UbidotsException

Methods: 0.00% ( 0/ 1) Lines: 0.00% ( 0/ 1)

DPR\API\Domain\User\User

Methods: 100.00% ( 6/ 6) Lines: 100.00% ( 12/ 12)

DPR\API\Domain\User\UserNotFoundException

Methods: ( 0/ 0) Lines: ( 0/ 0)

DPR\API\Infrastructure\Persistence\DAO\AuthenticationDAO

Methods: 0.00% ( 0/ 4) Lines: 0.00% ( 0/ 41)

DPR\API\Infrastructure\Persistence\DAO\BrandDAO

Methods: 0.00% ( 0/ 5) Lines: 0.00% ( 0/ 39)

DPR\API\Infrastructure\Persistence\DAO\CounterRuleDAO

Methods: 0.00% ( 0/ 6) Lines: 0.00% ( 0/ 49)

DPR\API\Infrastructure\Persistence\DAO\DAO

Methods: 0.00% ( 0/ 1) Lines: 0.00% ( 0/ 1)

DPR\API\Infrastructure\Persistence\DAO\DAOFactory

Methods: 0.00% ( 0/12) Lines: 0.00% ( 0/ 13)

DPR\API\Infrastructure\Persistence\DAO\DeviceDAO

Methods: 0.00% ( 0/ 8) Lines: 0.00% ( 0/ 99)

DPR\API\Infrastructure\Persistence\DAO\FunctionDAO

Methods: 0.00% ( 0/ 5) Lines: 0.00% ( 0/ 38)

DPR\API\Infrastructure\Persistence\DAO\MethodDAO

Methods: 0.00% ( 0/ 5) Lines: 0.00% ( 0/ 39)

DPR\API\Infrastructure\Persistence\DAO\ModelDAO

Methods: 0.00% ( 0/ 5) Lines: 0.00% ( 0/ 39)

DPR\API\Infrastructure\Persistence\DAO\ParkDAO

Methods: 0.00% ( 0/ 3) Lines: 0.00% ( 0/ 23)

DPR\API\Infrastructure\Persistence\DAO\TypeDAO

Methods: 0.00% ( 0/ 5) Lines: 0.00% ( 0/ 39)

DPR\API\Infrastructure\Persistence\DAO\VisitDAO

Methods: 0.00% ( 0/ 9) Lines: 0.00% ( 0/ 85)

DPR\API\Infrastructure\Persistence\DB

Methods: 0.00% ( 0/ 2) Lines: 0.00% ( 0/ 8)

DPR\API\Infrastructure\Persistence\DBException

Methods: 0.00% ( 0/ 1) Lines: 0.00% ( 0/ 1)

DPR\API\Infrastructure\Persistence\DBPDO

Methods: 0.00% ( 0/15) Lines: 0.00% ( 0/ 61)

DPR\API\Infrastructure\Persistence\DBPool

Methods: 0.00% ( 0/ 4) Lines: 0.00% ( 0/ 16)

DPR\API\Infrastructure\Persistence\User\InMemoryUserRepository

Methods: 100.00% ( 3/ 3) Lines: 100.00% ( 10/ 10)

**Results**

As mentioned above, our coverage summary does not provide the best estimation of our actual coverage for a couple reasons. Firstly, the unified backend contains some extra files not relevant to the visitation application and middleware authentication specifically, which lowers our coverage since they are not being tested and interfere with our coverage output. Also, we decided to mock the DAOs instead of testing them and worrying about the database state based on feedback from our advisors which also lowers our coverage. These files aren’t tested automatically but we are sure they work based on extensive Postman testing. Instead, what we do have tested, namely the Action and Model classes, both have high coverage. We have successfully written unit tests for all main and alternative flows through these classes which generally pass. The tests that don’t pass are mainly involving the error flows and involve issues with our testing tools rather than the files themselves. Overall, I would say this means we reached our designed threshold for testing and coverage and verified the proper behavior of all of the backend files used by the visitation application.

## **Acceptance Testing**

Preconditions for all tests: The ‘Testing Environment Dataset’ list of users has been updated and the queries in db/db\_schema/ncparks\_test.sql have been run.

Test Plan Executed: 12/09/2022

Results

| **Test ID** | **Description** | **Expected Results** | **Actual Results** |
| --- | --- | --- | --- |
| LoginUser | Precondition:   * User is on the Visitation login page.   Steps:   1. User enters “permission-1’ for username 2. User enters “perm-1!sdc06NCPARKS2’ for password 3. User clicks the “login” button | 3. The Visitation Dashboard appears with WIUM displayed. | Test passes without issue |
| LoginAdmin | Precondition:   * User is on the Visitation login page.   Steps:   1. User enters “permission-4’ for username 2. User enters “perm-4!sdc06NCPARKS2’ for password 3. User clicks the “login” button | 3. The Visitation Dashboard appears with WIUM displayed. | Test passes however it shows WIUM by default |
| LoginError | Precondition:   * User is on the Visitation login page.   Steps:   1. User enters “permission-2’ for username 2. User enters “fakepassword for password 3. User clicks the “login” button | 3. The login screen doesn’t update and the form stays filled | Test passes without issue |
| ViewDevicesUser | Precondition:   * Test LoginUser has passed * User is on dashboard   Steps:   1. User clicks the “Devices” button 2. User clicks the park dropdown and selects “All Parks” | 1. The system displays a list of IoT devices located in WIUM  2. The system displays a list of IoT devices from all parks | Test passes however it doesn’t filter to just WIUM |
| ViewDevicesAdmin | Precondition:   * Test LoginAdmin has passed * User is on dashboard   Steps:   1. User clicks the “Devices” button | 1. The system displays an unfiltered list of all devices from all parks | Test passes without issue |
| AddDevicePressure | Precondition:   * Test LoginAdmin has passed and user is on dashboard   Steps:   1. User clicks the “Manage IoT Devices” button 2. User clicks the “Add Device” button 3. User enters the following device information:    1. Name: pressure\_sensor    2. ID: 10002    3. Park: William B. Umstead State Park    4. Multiplier: 1    5. Function: traffic    6. Type: pressure    7. Method: Automatic    8. Model: Particle MQTT    9. Brand: Particle    10. Latitude: 35.8895    11. Longitude: 78.7518 4. User clicks the “Submit” button 5. User clicks the “Back” button | 2. The Add Device page appears  4. A “Connection estsblished” and “Device successfully added” message appears  5. The pressure sensor appears on the list of devices | Test passes without issue |
| AddDeviceBodyheat | Precondition:   * Test LoginAdmin has passed * User is on dashboard   Steps:   1. User clicks the “Devices” button 2. User clicks the “Add Device” button 3. User enters the following device information:    1. Name: bodyheat\_sensor    2. Number: 10001    3. SeeInsights ID: 10001    4. Multiplier: 1    5. Park: Bald Head Island State Natural Area    6. Function: Traffic    7. Type: Pneumatic    8. Method: Manual    9. Model: Particle MQTT    10. Brand: Particle    11. Latitude: 33.8737    12. Longitude: 78.0015 4. User clicks the “Submit” button 5. User clicks the “Back” button on the form | 2. The Add Device page appears  4. A “Device Created” message appears and the entered info is displayed  5. The body heat sensor appears on the list of devices | Test passes without issue |
| AddRuleBodyheat | Precondition:   * Test LoginAdmin has passed * Test AddDeviceBodyheat has passed * User is on dashboard   Steps:   1. User clicks the “Devices” button 2. User clicks the “bodyheat\_sensor” device 3. User clicks the “Add Counter Rules” button 4. User enters the following information:    1. Multiplier: 3.0    2. Date Range: September 01 - December 30 5. User clicks the “Save” button | 2. The “bodyheat\_sensor” information page is displayed  3. The Counter Rule creation form appears  5. A“Rule Created” message appears and the entered info is displayed | Test passes without issue |
| ViewRulesBodyheat | Precondition:   * Test LoginAdmin has passed * Test AddDeviceBodyheat has passed * User is on dashboard   Steps:   1. User clicks the “Devices” button 2. User clicks the “bodyheat\_sensor” device | 2. The View Counter Rules table displays 1 rule with start date 09-01 and end date 12-30 | Test passes without issue |
| EditRuleBodyheat | Precondition:   * Test LoginAdmin has passed * Test AddDeviceBodyheat has passed * User is on “bodyheat\_sensor” page   Steps:   1. User clicks the rule on the table 2. User enters the following updated information:    1. Rule Start: 06-01    2. Rule End: 12-30    3. Multiplier: 2.5 3. User clicks the “Save” button 4. User clicks the “Back” button on the form 5. User clicks the “bodyheat\_sensor” device | 1. The edit page is prefilled with the existing values for the rule  5. The View Counter Rules table displays 1 rule with start date 09-01 and end date 12-30 | Test passes without issue |
| DeleteRuleBodyheat | Precondition:   * Test LoginAdmin has passed * Test AddDeviceBodyheat has passed * User is on “bodyheat\_sensor” page   Steps:   1. User clicks the rule on the table 2. User clicks the “Delete Rule” button | 1. The edit page is prefilled with the existing values for the rule  5. The View Counter Rules table is empty | Test passes without issue |
| AddDataBodyheat | Precondition:   * Test LoginAdmin has passed * Test AddDeviceBodyheat has passed * User is on “bodyheat\_sensor” page   Steps:   1. User clicks the “Add Visitation Data” button 2. User enters the following visitation data :    1. Park: Bald Head Island State Natural Area    2. Device: bodyheat\_sensor    3. Raw Count: 10    4. Comments:    5. Timestamp: November 13, 2022 7:00 PM 3. User clicks the “Submit” button | 1. The Manual Data Entry page appears  3. The View Visitation Data table displays 1 visit on 2022-11-13 | Test passes without issue |
| EditDeviceBodyheat | Precondition:   * Test LoginAdmin has passed * Test AddDeviceBodyheat has passed * User is on “bodyheat\_sensor” page   Steps:   1. User clicks the “Edit Device” button 2. User enters the following updated information:    1. Name: bodyheat\_sensor2    2. Park: WIUM 3. User clicks the “Save” button 4. User clicks the “Devices” button | 1. The Edit Device page is populated with the bodyheat sensor values  4. The body heat sensor appears on the list of devices with updated fields | Test passes without issue |
| DeleteDevicePressure | Precondition:   * Test LoginAdmin has passed * Test AddDeviceBodyheat has passed * User is on dashboard   Steps:   1. User clicks the “Devices” button 2. User clicks the “pressure\_sensor” device 3. User clicks the “Delete Device” button 4. User clicks the “Delete” button | 3. A confirmation menu pops up asking to confirm deletion  4. The pressure sensor is removed from the list of devices | Test passes without issue |
| ViewDataBodyheat | Precondition:   * Test LoginAdmin has passed * Tests AddDataBodyheat and AddRule Bodyheat has passed * User is on dashboard   Steps:   1. User clicks the “Devices” button 2. User clicks the “pressure\_sensor” device | 2. The View Visitation Data table displays 1 visit on 2022-11-13 with a calculated count of 30 | Test passes however the raw/calculated counts are both 10 |
| EditData1 | Precondition:   * Test LoginAdmin has passed * Test AddDataBodyheat has passed * User is on “bodyheat\_sensor” page   Steps:   1. User clicks the 11/13/2022 timestamp row 2. User changes calculated count to 25 3. User clicks the “Save” button 4. User clicks the “Devices” button 5. User clicks the “bodyheat\_sensor” device | 1. The Edit Data page is populated with the bodyheat sensor visit values  5. The View Visitation table is updated to have visit with timestamp 11/13/2022, raw count of 10, and calculated count of 25 | Test passes without issue |
| DeleteData1 | Precondition:   * Test LoginAdmin has passed * Test AddDataBodyheat has passed * User is on “bodyheat\_sensor” page   Steps:   1. User clicks the 11/13/2022 timestamp row 2. User clicks the “Delete Data” button 3. User clicks the “Delete” button 4. User clicks the “Devices” button 5. User clicks the “bodyheat\_sensor” device | 2. A confirmation menu pops up asking to confirm deletion  5. The View Visitation table is empty | Test passes without issue |
| ViewDashboard1 | Precondition:   * Test LoginAdmin has passed * User is on dashboard   Steps:   1. User views the dashboard on the Visitation landing page 2. User selects the parks dropdown and selects “Bald Head Island State Natural Area” | 1. The dashboard displays graphs of visitation data for William B. Umstead State Park over time  2. The dashboard is updated to display the graph of visitation data for Bald Head Island State Natural Area over time | Test passes without issue |
| ViewDashboard2 | Precondition:   * Test LoginAdmin has passed * User is on dashboard   Steps:   1. User views the dashboard on the Visitation landing page 2. User selects the viewing data dropdowns and enters these dates    1. From: 2018    2. To: 2019 | 1. The dashboard displays graphs of visitation data for William B. Umstead State Park over time  2. The dashboard is update to display older 2018 data with 0 total logs | Test passes without issue |
| ViewMetrics1 | Precondition:   * Test LoginAdmin has passed * User is on dashboard   Steps:   1. User views the comparison metrics on the Visitation landing page below the dashboard | 1. The comparison metrics section displays “Total yearly visitors”, “Compare to the same date range”, and “Total Park Visitors” charts for William B. Umstead State Park | Test passes without issue |

# **Suggestions for Future Teams**

*Author(s): Shruti Jadhav*

*Reviewer(s)/Editor(s): Quentin Sieredzki*

## **Additional Visitation Functionality**

One of the requirements included making a new and improved dashboard. Our final implementation includes one main line graph to see visitation over time and a couple of metric boxes to compare trends from the past. This dashboard definitely has room for extra features of all sorts. The sponsors may like this to be implemented in more detail for the future if this specific application is still being worked on.

Additionally, another requirement that we started but could be expanded upon is user permissions. The sponsors mentioned that they were redoing these permissions and not to focus on them at the start of the project, so we did not focus on them very much until later in the semester. Because of this, the implementation that we have for the user permissions is a little more rudimentary than the rest of the code and only restricts permissions for adding/editing/deleting devices and counter rules, and could be improved in the future as the user permissions are better defined for the rest of the applications.

Also, with viewing our visitation data, we were unable to implement a full pagination fix in time which would be useful to look into. While the visitation data table still loads, albeit slowly, we implemented this fix by increasing the maximum memory of all Action classes to 2GB which is only a temporary fix. Optimally, the frontend tables would be reconfigured to make new API calls for each page which pass in the number of rows per page and the requested page.

**Unit Testing**

Another area for improvement is in our unit testing. Some bugs exist in our backend PHPUnit tests, especially on the error flows, which could be fixed for the sake of best practices, even though we are pretty sure that all of the implemented functionality is working. Adding on to this, PHPUnit could be additionally configured to automatically ignore the untestable files which were previously mentioned. Both of these would provide useful improvement to the readability of the coverage report as well as create a better framework for building future tests off of. Additionally, future developers could look into expanding the unit testing onto the frontend React implementation using a tool like Jest so that we can automate frontend testing.

## **Future Remakes**

There are many applications that are maintained by the NCDPR, and remaking one or several of these applications could be a suitable task for a future project. The sponsors mentioned that they had a lot of project ideas and applications that the senior design center could help them out with. One specific project that was talked about was the inventory tracking system.

## **Developing Next Project**

The containerization done in this project allows for additional apps to be developed separately from the rest of the system. This means that the next team to take this project will have the freedom to choose whatever technologies they and the sponsors agree on for re-making new applications, or whatever other task they may decide to do.

The legacy database is large and unwieldy, with many tables containing columns of information that may be irrelevant to the function the table serves. The legacy database must remain in use for the remaining applications; however, the sponsors are open to completely overhauling the database as well as just adding new tables and schema for each new application revamp.

# 

# **References**

[1] Anon. 2011. Unsupported branches. (2011). Retrieved September 27, 2022 from https://www.php.net/eol.php

[2] Cole Goodnight, Kyle Manahan, Matt Murdoch, Josh Roddy, Colin Szatkowski, Justin Wald. 2022. Next Gen System Sustainability Interim Project Report. Retrieved on 9 September 2022.

[3] Anon. Hash\_equals. Retrieved December 9, 2022 from https://www.php.net/manual/en/function.hash-equals.php

[4] Anon. 2022.(March 2022). Retrieved October 7, 2022 from https://www.slimframework.com/