DPR Visitation Application Dev Guide

Updated Application with Unified Authentication - Fall 2022

This document focuses on aspects of the project of interest to developers continuing our work, and includes instructions such as when to update the development environment setup and use, or how to make modifications and extensions to the project. It includes all commands required to install development tools and dependencies on a given operating system, run the software from a checkout, and run all tests.

# User Level Logins

In the current implementation of the project, we are using a series of temporary user logins each with one of the possible roles on the Visitation application. These were the only users we had access to, but they may continue to be useful for development and testing purposes without having to rely on real user login information. The roles are no access, base level, manager, admin, and super admin which go from permission level 0 to 4 respectively.

No Access User

Username: permission-0

Password: perm-0!sdc06NCPARKS2

Base Level User

Username: permission-1

Password: perm-1!sdc06NCPARKS2

Manager User

Username: permission-2

Password: perm-2!sdc06NCPARKS2

Admin User

Username: permission-3

Password: perm-3!sdc06NCPARKS2

Super Admin User

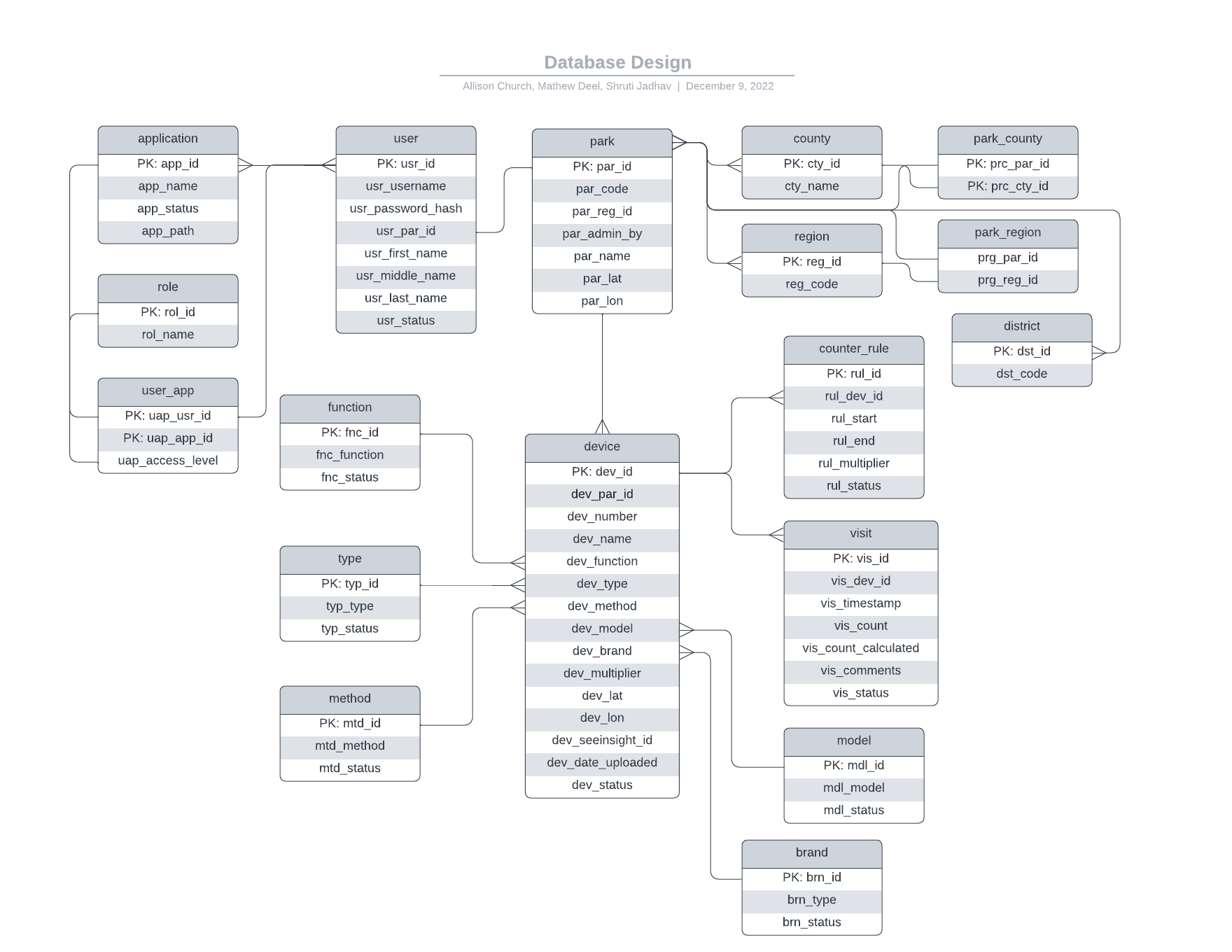
Username: permission-4

Password: perm-4!sdc06NCPARKS2

# Database Design

This is the database for the Visitation application as well as the unified backend which will expand off of this. At the top level is *User* which has a join table with *Application* and *Role* which is used to verify user permissions for all updated applications. Building off of that, a user can have a *Park* with associated *County*, *Region*, and *District* information. Unique to our Visitation application, *Device* tracks all of the devices associated with parks and can have multiple *Counter Rule* and *Visit* instances associated with it. Lastly, The *Function*, *Type*, *Method*, *Model*, and *Brand* tables link to many devices in order to minimize redundancy.

The Visitation application does not have functionality for adding users, as this is a system wide process outside the scope of our project. Users that are able to log into the application must already exist inside that *User* table.



# Frontend

The frontend of the Visitation application is written in React JS. We use react-router to navigate around the webpage, MaterialUI for styling our components, and ApexCharts for our dashboard visualizations. This section will be split into several subsections: App Structure, MaterialUI, API Calls, and testing.

App Structure and Navigation

The root directory for the frontend is visitation/. It is here that any npm install commands should be run to install new dependencies. Conversely, Docker will automatically run npm to install the dependencies specified in the package.json file so that npm should never have to be manually run unless an issue occurs. Inside this directory is visitation/src/App.js, which is the root for the frontend components in this application. App.js contains all of the routes for the application, which are defined inside route components. Each route component contains a path, which is the path inside the application, and an element, which is the component that path goes to. Note that the root path (‘/’) is set to the login page, so going to the base path (in this case localhost/visitation) takes you straight to the login page. Once a user logs in, they are redirected to the dashboard landing page regardless of their role stored in the database. These frontend pages contain buttons, which when clicked call the navigate function to navigate to the correct path in the application. Aside from App.js, all of the frontend pages such as the login page are stored in visitation/src/Components in organized folders. There is also visitation/src/utils/api mentioned later which is used to define the API endpoints and their routes.

MaterialUI

Rather than styling our pages with CSS, we are using MaterialUI to provide pre-styled basic components for us to use. These components include Button, Select, TextField, Autocomplete and others. Components are organized on the front end in a flexible Grid structure, which creates columns and defines spacing across the page that will adjust to the window size. Boxes are used within the grid to provide padding and margins around a component, a group of components, or other Boxes.

API Calls

All calls to the backend are handled through Axios using an HTTPClient.js class. Once the API endpoint has been added to the relevant service class in visitation/utils/api, you will be able to easily make API calls in the frontend components. For example, if you were creating a visit, you would call APIClient.visit.createVisit(1, visit) and this would link to the createVisit method in visitation/utils/api/VisitService.js. As this travels to the backend, api/src/Application/Routes/VisitationRoutes.php will route to the appropriate Action class which would be api/src/Application/Actions/Visitation/Visits/CreateVisit.php in this example. This will do any necessary preprocessing actions before calling the DAO class which will handle it. Here, api/src/Infrastructure/DAO/VisitDAO.php will call createVisit(Models\Visit $Visit) and it will be added to the database. Authentication is handled through middleware so this will not require anything to be added to the body of the API call. Back on the frontend, it will wait for the response, convert the response to json, and then take the data it receives to do the desired action

Testing

Currently, our frontend components are not set up to support unit testing frameworks like Jest which would be a useful addition to this project. Instead, we are unit testing our backend using PHPUnit. All of the tests for the backend are located inside the api/tests directory and are divided into subfolders based on whether they are testing the Model classes or the Actions. Our DAOs are purposefully not being tested and our Action tests instead use DAO mocking. To run all the tests, navigate to the root directory of the repo while docker is running, and then run the following commands:

1. Run the command:

docker ps

1. You will see a list of all the containers, copy the container ID of 2022fallteam06-nc-parks-2-api
2. Bash into the container:

docker exec -it <container ID> bash

1. In the root, run this command to test:

./vendor/bin/phpunit --verbose tests --coverage-text

The above command runs all the unit tests but you can specify what directory of tests you want to run.

Note docker does *not* need to be rebuilt every time you write more tests and rerun.

# Stack Networking

## NGINX Reverse Proxy

The NGINX reverse proxy redirects requests to the DPR stack to the appropriate container in the stack based on the request’s URL. This allows functionality to be switched between containers (such as when a legacy app is rewritten) with the change of a couple lines in the NGINX config.

The reverse proxy also provides the service of rewriting requests to remove path prefixes such as /api/dprcal/ to make containers ignorant of their position behind a reverse proxy.

It’s important to note that should any change be desired in the routing of a request within the stack, the nginx config will be the primary method of doing so.

# Project Orchestration

## Docker Compose

Docker Compose is the backbone of the new DPR stack. It manages the building, configuration, and running of all of the application containers. A description of containers and Docker Compose is out of the scope of this document so please follow these links for details:

* <https://docs.docker.com/get-started/>
* <https://docs.docker.com/get-started/08_using_compose/>

## Secrets

Any piece of sensitive configuration information is considered a “secret.” Examples of secrets are the encryption keys for https, database admin passwords, etc. The proper handling of these secrets is imperative for application security. Luckily, Docker Compose provides a

handy method for inserting these secrets into our application at runtime. The DPR stack has adopted the convention of application secrets being declared in plain text in files in the secrets directory of the project. On runtime, the path to these secrets are identified by environment variables and loaded in by Compose. Once read in, Compose copies them into a folder in the appropriate containers with appropriate permissions as specified in the docker-compose.yaml. From there, the applications that need them can access them as regular files with minimal permissions.

For more information, see <https://docs.docker.com/compose/compose-file/#secrets>.

## Environment Variables

There are several points in the Docker Compose file where parameters are specified with environment variables. These are intended to be specified primarily in a .env file, filled with key-value pairs, stored in the envs/ folder. That said, any parameter that is also specified in the shell will overwrite the corresponding key in the .env file. Any environment variables not specified in either the shell or the .env specified with –env-file cmd arg will be set to an empty string resulting in unknown behavior.

It is recommended that any parameter that should be specified at runtime is specified using an environment variable with a description in example.env in the envs directory.

Settings/Dependencies

Within the api/app directory of the NCDPR file system, settings.php references the environment variables above and creates a global Settings object that configures the logger, database, and ubidots settings that will be referenced by the applications.

return function (ContainerBuilder $containerBuilder) {

// Global Settings Object

$containerBuilder->addDefinitions([

SettingsInterface::class => function () {

return new Settings([

'displayErrorDetails' => true, // Should be set to false in production

'logError' => true,

'logErrorDetails' => true,

'logger' => [

'name' => 'DPR-API',

'path' => isset($\_ENV['docker']) ? 'php://stdout' : *\_\_DIR\_\_* . '/../logs/app.log',

'level' => Logger::*DEBUG*,

],

'database' => [

'engine' => getenv('DB\_ENGINE') ?? null,

'host' => getenv('DB\_HOST') ?? null,

'port' => getenv('DB\_PORT') ?? null,

'dbname' => getenv('DB\_NAME') ?? null,

'user' => getenv('DB\_USER') ?? null,

'password' => getenv('DB\_PASSWORD') ?? null,

'charset' => getenv('DB\_CHARSET') ?? null,

],

'ubidots' => [

'token' => getenv('UBIDOTS\_TOKEN') ?? '',

'base\_url' => getenv('UBIDOTS\_BASE\_URL') ?? ''

]

]);

}

]);

};

When building the container, dependencies are configured within dependencies.php, which creates classes that access a particular group (or groups) of settings.php settings information.

return function (ContainerBuilder $containerBuilder) {

$containerBuilder->addDefinitions([

LoggerInterface::class => function (ContainerInterface $c) {

$settings = $c->get(SettingsInterface::class);

$loggerSettings = $settings->get('logger');

$logger = new Logger($loggerSettings['name']);

$processor = new UidProcessor();

$logger->pushProcessor($processor);

$handler = new StreamHandler($loggerSettings['path'], $loggerSettings['level']);

$logger->pushHandler($handler);

return $logger;

},

DBPool::class => function(ContainerInterface $c) {

$settings = $c->get(SettingsInterface::class);

$dbSettings = $settings->get('database');

return new DBPool($dbSettings);

},

UbidotsAPI::class => function(ContainerInterface $c) {

$settings = $c->get(SettingsInterface::class);

$ubidotsSettings = $settings->get('ubidots');

return new UbidotsAPI($ubidotsSettings);

},

]);

};

This fluid abstraction of settings allows for a quick sort of ‘top-down’ change in the system’s settings relatively simply by changing the parameters of its environment file or using a different one. Additionally, project server settings can be changed easily through editing the nginx default.conf.template file.

# API Documentation

If you are interested in learning about the API we have implemented, the documentation describing all of the REST API endpoints is contained in the Final Project Report document. Our new API is part of the unified backend conversion and is set up to facilitate expansion as needed to accommodate new applications. We currently have multiple groups set up in api/src/Application/Routes with *AuthRoutes* being used for authentication. ParkRoutes.php groups some of the shared functionality involving parks. Lastly, we have groups set up for each of the apps such as CalendarRoutes.php, FilesRoutes.php, and VisitationRoutes.php. Our visitation application further breaks this down into groups for devices, brands, functions, models, methods, types, counter rules, visits, and Ubidots.

# Webhook Development

Alongside the application itself, developers will also need to be aware of the Ubidots webhook and how it works in order to understand how to update it. The Final Project Report covers the specifics of the design but whenever the IoT device infrastructure is expanded, one of two things will need to be done on the SeeInsights portal. If a new device is being added to an already configured park, that device will need to be added to the park’s device group. Otherwise, if you are configuring a new park from scratch, a new device group will need to be created which contains all of the relevant devices. This group’s webhook will work similarly to others in that if any hourly counts are above zero, it will make a POST request to the server at api/visitation/incoming. Our server is currently an NCSU hosted virtual machine however, this will need to be swapped out for actual deployment.