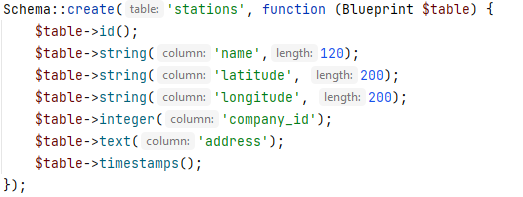
# NanoCHARGER API

The following document describes the implementation process of the API that is responsible for concatenating data based on coordinates and radius.

**Laravel Installation and Pre-Development Steps**

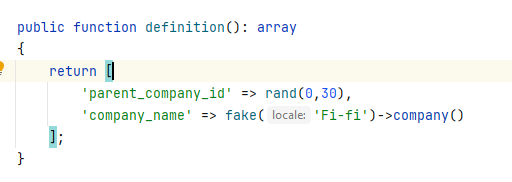
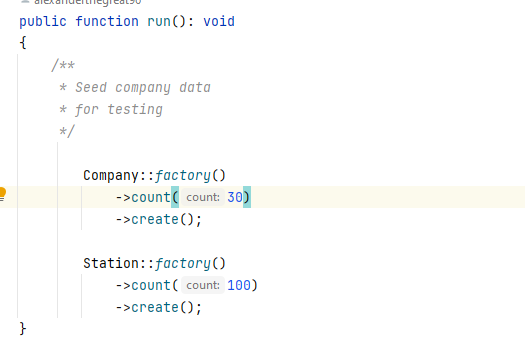
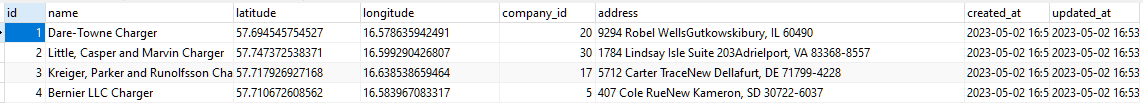
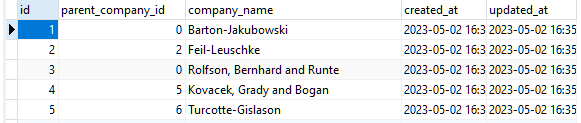
Steps:

* Create a new Laravel installation using composer:
  + Composer create-project laravel/laravel nanocharger-api
* Define de containers for the API:
  + Nginx container for the API Gateway / Web Server
  + MySQL Container for the database
  + Redis Container for caching
* Create the database migration based on the required table structure:
  + php artisan make:migration create\_stations\_table
    - 
  + php artisan make:migration create\_companies\_table
    - 
* We cannot forget to version everything. When developing APIs and Microservice solutions (which are smaller APIs), we have to make sure that we can version the API. The versioning is done simply by storing and prefixing our controllers, models, factories and routes using a structure such as: /api/v1

This is extremely important because as things progress with our application, our customers, or even our apps may be using the previous API version and transitioning it to the current one might take time. Versioning the API will ensure that the old endpoints still work, while the new ones may be in development or in a beta stage and when the time comes people can make the transition to the new one. With that out of the way, I went ahead and generated the models using the said prefix:

php artisan make:model Api/v1/Company

php artisan make:model Api/v1/Station

* Seed the databases with data so we can play with. For this example I’ve used the built in eloquent model factory that allows me to seed database with records:
  + Generated the boilerplate code using:
    - php artisan make:factory Api/v1/CompanyFactory
    - php artisan make:factory Api/v1/StationFactory
  + The first step was to populate the companies table, this was done using the code below:
    - 
  + The second step was seeding the stations and here things are a bit tricky. Essentially, what I did was to pick a random point in Google Maps, a city in Finland ,for this example and then used a function that another smart guy wrote, to generate random points along a radius in kilometers (the function will be in the git repo, so you guys can see what it actually does):
    - 
  + The final step was to define how many records I wanted inserted for both factories right before running the seed command. This is done in database/seeders/DatabaseSeeder.php
    - 
  + Final step was to run the seed command which is php artisan seed
* The final result looked like this:
  + 
  + 

**Application development**

We reached the step where can actually implement the functionalities required. We have plenty of records to test our application flows.

The obvious thing to do here was to implement the relationships. These are extremely important because we want to access data in a recursive way starting from the root companies, all the way to the last children company, of course, chargers too. This is done by defining a few relationships.

Whin the Company Model I fined a few relationships like so:



What this allows us to do is to be able to query all the ids that fall under a certain company and based on these ids we can run our stations query. Additionally, I implemented some helpers that aid in that process. 

The function above flattens the array and give us a structure that we can use to run our queries.

With the models out of the way, I went ahead and handled the API authentication, as we have to protect the routes with some kind of authentication method. For this example, Sanctum Authentication (which comes by default with Laravel 10), was used. Sure, any other kind of middleware handler can be used, but for this example it should do just fine.

I went ahead and created 2 new routes to handle the user registration and authentication (although the authentication is not used as we are only using the token we just grabbed).

I mapped 2 routes which we are using for this:

* “/api/auth/create” -> which handles user registration
* “/api/auth/login” -> which handles user login

I used artisan to generate the controllers and requests: (we use the FormRequest implementation to be able to validate form inputs basically)

* “php artisan make:controller Api/AuthController”
* “php artisan make:request Api/CreateUserRequest”
* “php artisan make:request Api/LoginUserRequest”

As I mentioned before, the requests are simply validating the inputs that the client is sending to the server.

Once a post request is sent to “/api/auth/create”, the server will respond with an array telling us that the user has been created and that the generated token is returned.

## Locating stations

In order to locate stations we can implement various methods. I’ve made multiple, not sure that was the best idea, but I tried some methods that involved using the web server (PHP) to sort the data and others that rely only on the MySQL Server. The general idea here is to keep an eye out for how server scaling works. Normally, database server’ CPUs do not scale as much as Web Server CPUs.

It’s worth noting that I’ve implemented a scope within the Station Eloquent Model. A scope is basically a query modifier used for sorting data with certain things. 

scopeClosesetToThis takes a few parameters: latitude, longitude, unit and radius and then uses them to triangulate data based on Haversine Ecuation.



The implementation above uses purely the database server to pull these locations. I’ve achieved the same result using PHP only.



The next method implemented is grouping stations with the same location together and sorts them in the same way , from the closest to the farthest away.



The reorganizing according to distance is done by using Remap::organize() method. I came up with the idea of base64 encoding the latitude and longitude together. Hence multiple stations will have the same coordinates, we can use that to filter the array. Is it the smartest idea? Not really, we can most likely do better, does it work, hell yeah!

The exact same result is replicate below using pure PHP for computing distances and organizing data. 

Now, getting back to your request. Which is pull all children companies stations and group them together by distance .

The method that I though would suit this without getting into insantely complicated custom recusrive queries, so that we don’t use too much MySQL for processing and at the same time we can still keep the power of Eloquent in case we need to use additioonal scopes or relationships. I pulled all the company ids recursivesly based on the company\_id input then wen to pull data for stations.





Same result achieved with PHP to compute data.



You might be asking yourselves why I built so many methods. What do we need them for? We can say performance testing and overall data manipulation. Maybe we want all the stations nearby, maybe we want only stations for a certain company. I don’t know. I just wanted to play with these things since I haven’t worked with coordinates at this level before. This was a really fun experience, if I might add.

Now these requests will produce results that will have stations grouped according to distance as well as grouped according to companies recursively. While I am aware that the code can be improved a lot, I really hope that some of the methods of querying data (without killing database servers) are alright.

It’s worth mentioning that I am using Form Requests to make sure that keep post data and request data in place without doing redundant checks. The methods for handling REST requests are described in the API documentation. They basically allow you to perform CRUD operations of stations and companies.

An additional feature that I will implement if time servers is caching the requests using Redis. My method of approaching this is computing distance between a previous location and current location and checking if we are within a 50 meters and to serve data from the cache rather than computing it again.