User Management system development report

Architecture

The system has a layered architecture, as it was implied by the instructions themselves. There's the database, the API that operates with the database (and returns results), and the frontend project, all separate from one to another.

When creating the database with Laravel's migration command, this would create a bunch of tables, including a table called "users", this is Laravel's default table so it was decided to ignore it and create a whole separate table for the system, one called user_registries. The API backend application has a model called UserRegistry and this is controlled by a HTTP handler called UserRegistryControler.

To route the requests for the API, and for major security, also as requested, the project uses JWT for authentication, handling the validation of pre-processing requests, making sure that only opened sessions of authenticated users can execute these APIed methods. Naturally this does not really include login and register as those two operations are usually done before a session has opened as part of their function is to create/open a session.

The selected database was MySQL as it is part of XAMPP, a tool that abstracts many tools into one, making the overall development more practical and quicker.

The frontend was, indeed, made with Vue.js and Vuetify within it, communicating to the database through the API's only if it has locally stored any session/access token. There are components that handle not only the page itself but also the navigation bar and the verification of the current session whenever any page other than login and register is opened, to return the user back to login if they're still not logged in.

Key decisions

Something I already knew but once again experienced first handed during development is that these technologies are updating often, outdating many information sources and solution alternatives to different issues. Sometimes for the better, sometimes for the worse. This also involves incompatibility with different versions, leading to having to find the compatible elements for each other.

This laptop had a hard drive but was replaced with a solid state, while the hard drive is now used as extra storage, this means that a lot of settings and variable environments were gone and I have not used them for some time, and while XAMPP covered plenty of the requirements for this system specifically, it did not cover everything. So, a key decision I was able to take was to see this as an opportunity to get out of my comfort zone and try learning these new updates and see the changes these frameworks have gone through.

Another key decision was keeping it simple but presentable, as I usually like modularity and keeping functions and layers properly encapsulated, avoid repeating code if it's possible to separate it into its own thing to be called by other scripts, but due to the time limit, I had to not follow this direction and not make the fanciest solution but have something that works properly still.

Test Driven Development approach

Because of the time limit and not being in the best scenario when it comes to my current daily life, I was not able to have a proper TDD approach, instead, the way this system was developed was as I'm used to write code, which is testing it myself instead of automating the process. I was not sure if I'd have everything done in time, so making automated tests for yet-to-polish functions in both backend and frontend did not seem like the best approach for this scenario.

Because of that, during development I was testing the APIs with the software Postman, sending the requests myself and seeing the results it'd give me, if I set the parameters correctly, etc.

I chose PEST over PHPUnit only to get to experience a different framework and it was very similar to PHPUnit despite how it is presented by basically everyone on the internet, I used the instructions in PESTs official website and, although I know it is based and made by the people behind PHPUnit, I was expecting it to me more different, "simpler" as many said. And yes, the examples I found online were different from the ones I ended up making, maybe this is another instance of some version inconsistencies of some sort.

After almost finishing the system, I made the test file and executed it, and I found a couple of errors, mostly typos and missing expected parameters in the URL. Thus, proving that they were useful at the end.

In the end, I wanted to make this test file as I'm used to making tests, which is what I sometimes call a "tour" through all the functions the system has, so the most obvious approach was to have the test create an account to log into and do the operation tests with it. The order is, specifically:

- Register a new account: testUser
- testUser logs out
- testUser logs in
- testUser gets all user registries
- testUser gets the data of the first user registry
- testUser adds a new account: anotherUser
- testUser edits anotherUser, changing the username to: anotherUserEdited
- testUser deletes anotherUserEdited
- testUser deletes itself

This list is also visible in the README file.