

Technical Report

First of all, I would like to thank the recruiting team for the attention and patience along throughout the technical assessment process, as well as, the time extensions that allowed me to finish most of the functionalities that I had planned on doing for each of the projects, and I would like to commend the assessments that were put together, they handle exceptionally a lot of the steps developers take during everyday job life.

This document aims to explain my thought process and development process I did during the timespan of the assessment, and to segregate this information from the README information from the repositories because its focal point is not report development choices and instead tell information about how to use the application present in the repository in question.

I will break down the information in three separate sections, which can be found in the table of contents below, general stuff which applies for both of the tests followed by test specific sections to report and summarize my decisions.

Contents

General Decisions	2
Backend (Thor)	3
Architecture	3
With more time I would	4
Frontend (Loki)	5
Architecture	5
With more time I would	6

General Decisions

With the opportunity of taking a full-stack technical assessment I thought it was a good idea to toy around with the idea and create an imaginary company in which each of the tests is an individual application, this was fun and helpful, because it made me have a clear vision of how the flows would work and who would use the applications, which made me take some decisions based on this scenario and helped me architecture the systems accordingly.

For this imaginary company I created a GitHub organization to store, centralize and organize the codes in a tidy manner, and also, write cohesive documentation to back the application's technical decisions of this imaginary company. On the GIT topic, on a real work environment I would go for the Pull Requests and Rebasing approach, so that the repositories would have a clear git history to identify where changes came from without compromising the git flow from the master branch, but as I had a tight deadline based on some factors I will bring up I choose to push every change to the master branch in a way that almost every commit is a stable version of the projects, this made me speed up the development process a bit.

Going with the vision of this imaginary company and that both of the assessments would be real life applications, I updated the expected data models of both the applications so that they would meet the required needs, this made me deviate from the original path but I strongly believe it was a good decision, because by doing so, I was able to display some of my infrastructure and business rules knowledge.

Now for not so important stuff, the theming. As it was an imaginary company, I had the freedom to toy around with names and I decided to go with the cliché approach of Nordic Gods, which I think does the trick well. I started with Thor for the backend, because he is strong and should be the backbone of the flow, then I followed with Odin for the database, the all-father, the all-seeing and the all-knowing, titles that give a clear figure of a database that holds all the information. Loki came soon after, the trick God that toys around by using illusions and visual trickery, just like a website that displays visual information for the user and hides a lot of work behind the curtains. And lastly the network name that came naturally after the other names, Asgard, the place where the three gods reside.

Backend (Thor)

My PHP knowledge was worse than I thought at the beginning of this test and my Laravel knowledge was practically inexistent, so I researched a ton before starting my works, starting by the Laravel API, which I found very overwhelming at first and lackluster in some topics, like API description of most functions and what they did and what they could do. This made me go after some third-party documentations at first to get a feel of the framework and how to behave myself when developing in it.

The Laravel start project was also overwhelming, it had a lot of files which I didn't have a single clue what did, which made me go after some repositories with less code so that I could understand it better with the dilution approach, and by doing so I ended starting up with a mix of Laravel's built in initial project structure and [this structure](#) by @gentritabazi on GitHub. I also followed [this GitHub repository](#) by @alexeymezenin which has a very in-depth explanation of Laravel's good practices.

Architecture

The first thing I did was to try and move the Laravel files to an `"/api"` folder to reduce the amount of folders and files on the root of the repository, but I quickly gave up on the idea because I ended up having a lot of trouble with artisan and composer's commands and then settled with changing only the structure on the `"/app"` folder organizing the file packages by their functionality (e.g. Asset, Vulnerability...) instead of by their type (e.g. Controllers, Services, Repositories...) except for the commonly used files across the application which ended up being stored in a new package called `"/infrastructure"`.

This project structure made it quite easy to identify and understand the routing system and where each endpoint went to handle it's functionality, which was very helpful when adding the few more routes for the frontend integration. These routes are "devices", "gateways" and "topology", the latter is singular because there'll be only one topology for each client, and in this case only one topology at all.

I'm most familiar with Spring's project structure and this made me wonder if I should create a Repository for each of the CRUD functionalities, which ended up not happening because of Laravel's Eloquent DataBase, which I found very cool by the way, much easier to use than the integrations we have in the Java ecosystem. Also, when developing with Spring the creation of a custom exception handler is always needed to format the response to match the desired output when some error happens,

so I ended up adding one to clean up the response that goes back to the user, preventing it to see the stacktrace and only what is really needed, which in this case is the exception message.

One thing I don't know if I did correctly when following the project structure I went with was with the use of Eloquent in the tests, seeing the ServiceTests handle database operations doesn't quite sit right to me. Talking about tests, faker being natively integrated with Laravel was quite helpful, I just had to adjust some things here and there, like the fake random float number because it was having some problems with the precision.

With more time I would:

- Add a rate limit per user on the API calls, to try and prevent DDoS attacks;
- Add a async job flow to fetch vulnerabilities from the NVD api based on the fields of any newly created Asset;
- Integrate with ClickHouse for aggregated queries, which I was not able to make work;
- Fix Asset and Vulnerability integration and service tests removing the need of hardcoded uids because RefreshDatabase kept states between tests;
- Upgrade test coverage and functionality extending it to the frontend specific CRUD functionalities;
- Change varying value string ids on frontend specific tables to the internal id and external id policy;
- Add CRUD operations for frontend specific tables that are read-only for now;

Frontend (Loki)

The frontend development was very smooth and I had a blast while doing so with NextJS and React, because of my prior Svelte and SvelteKit knowledge, which made me able to translate almost everything with some tweaks to the syntax and logic. Examples are components conventions, TypeScript integration and external files, and file/folder based Routing system.

Architecture

The first thing I did was go to the testing company's website to take an in depth look into the UI, color scheme and theme, searching and looking through all that is available, text placement, images, gifs and videos of internal tools, which gave me a good headstart on developing the UI.

Initially I had plans to use a semi-complete component library like Shadcn, to give me a big headstart on the development and to reduce the time it took to do so, but I quickly moved on from it because too much unnecessary code was generated and I could simply create my own components as the functionalities list was not that extensive. Except for the data-table component, which I imported from a library because of two reasons, first was that my time was already running low and I had to take an extension, and the second is that it attended all the required needs and had robust customizability support.

On the topic of components, there are quite a bit and most of them are not necessarily reusable, in these cases they are being exported from a package specific export system, to make them somewhat private to a specific page. One of the components I could have made generic with props would be the ReactFlow topology node component, which would receive props instead of having one custom component for each Node Type, but I opted for not doing so because if in the future one of the Nodes needs to be changed to have a specific UI that is not shared with the other ones it can be easily done.

The custom components will have their Loading/Empty/Error states located either inside themselves if the amount of functionalities is not too much, otherwise they will be located in a new component with similar naming convention and placed in the same package.

I also was able to learn and have hands-on experience with search parameters filtering, which was very fun and insightful, because it tunes up reproducibility and shareability of webpages in a way that prevents current displayed state. I ended up implementing a manual validation for the query filters, this with the fact that I did not add the optional module of notes made me not use the Formik dependency.

I will certainly take this filtering system along in my journey as a developer.

With more time I would:

- Add notes optional module;
- Add Create, Update and Delete operations for Assets, with optimistic UI, toasts to display important data and a button to revert last action;
- Create custom component for edge, to add custom styles and remove the need for the hacky conversion of edgeType from "edge" to "default";
- Add animations for component rendering and destruction, to ease the ins and outs;
- Fix Node component colors not appearing on the MiniMap;
- Change Icon library to unplugin-icons because of how the icon bundling works, reducing final application bundle size;
- Change Assets listing from table with pagination system to a endless scrolling table;
- Sync in real time the filtering input with the URL, to remove the need of a search button;
- Dark mode, definitely a Dark Mode to code at night;