Architecture

The solution is separated in multiple projects:

- **ClayDoorMain**: Contains the Program.cs to start the application. This is where the configuration happens.
- **ClayDoorController**: Contains the endpoints and associated middlewares.
- **ClayDoorModel**: Contains the business logic and the definitions of objects.
- **ClayDoorDatabase**: Contains the logic to access the database.
- **ClayDoorModel.Test**: Contains the Unit Tests for ClayDoorModel.

The goal of this architecture is to keep the business project (ClayDoorModel) pure. In other words, it should not depend on other projects and have as little dependencies as possible. The Model can describe the interfaces it needs, and thanks to Dependency Injection, the implementations can be given to the model.

Technologies used

- **AspNetCore**: There is no need to have a UI, and it is a standard solution to build a RESTful API.
- .**Net 8**: It is the latest LTS. If there is nothing preventing the use of the latest LTS, I don't see why not chose it!
- **MariaDB**: I had no experiences with that DBMS, but since it is described in the role description I'm applying, I wanted to have a first experience. If I had the time to implement integration tests, I would have given a look at SQLite for this part.
- As for **authentication**, the choice was made to have a JWT Bearer token. This is the most common solution when looking at how to implement auth in asp net and there is a lot of documentation on how to use and configure it, so it looked like a good choice for a first time.
- **Entity Framework** has been used to access to the database but it has not been used to create the database as I wanted to be sure to have control over it.
- **Swagger** is useful for the tests, as is **Postman**, both were used during the development.

Choices

- I saw that it is common to embed the claims of the authenticated user into the JWT token, but my issue was that if these claims change after a token is issued and before it reaches end of life, the user keeps its rights for a short time. So I decided to fetch the permissions of the user each time they are needed.
- For authentication, I followed a User-Role-Permission pattern (RBAC) as it is simple while staying scalable. If needed, it can be easily extended to provide more precise control later.
- The authentication only uses a username, this is to simplify the task, in a real environment, a username + encrypted hash of password could be used and compared to the hash stored in the database. Better solutions could be to use an external service to authenticate the users.
- Some important data are present in the config file, they should not be there but rather be retrieved using secure methods like Azure Key Vault.Entity Framework's entities are not mapped directly onto the model as I wanted this last to stay independent from the ORM.

State of the solution

The solution is available at https://github.com/delevoye977/test-task-clay/tree/main

And a demo at https://youtu.be/8CNG7Am87Mo

It is possible for a user to get a token and then the user can get the list of doors, try to unlock one of them, and have a look at the logs of doors if they have the right permissions.

What is missing:

- Some integration tests, to at least ensure that users can do the basic tasks
- Endpoints to manage (CRUD) users/roles/permissions/doors

I didn't have the time for these because I spent too much time trying to make a Generic CRUD Service + Repository as it looked like I was about to do very similar things a lot.

You can see the WIP branch on https://github.com/delevoye977/test-task-clay/tree/dev/doorCRUD

Look back at the task

Starting this kind of project from scratch was a first to me and I spent around 20h on the project, from design to implementation. If I had to do this project again, in a team situation:

- I would have spent as much time on listing the needs and designing the architecture.
- I would have then asked someone to check if they thought I was going in the right direction.
- I would have asked if a similar project already existed, and if someone could show it to me (it helps to understand the project but also to see what was well-thought or to rethink).
- Then using that project and the insights from colleagues, started implementing the solution.
- The authentication/authorization were time-consuming as I wanted to be sure to understand things I did as much as possible, maybe seeing a project already implementing it could have been useful.

Even if some aspects are still missing, this is a robust architecture which can be adapted to more situations in the future. I believe my solution works and can meet the requirements with a bit more of development.