CELITO NEST JS ASSESSMENT

This document is a quick description of the assessment done for the nest js project.

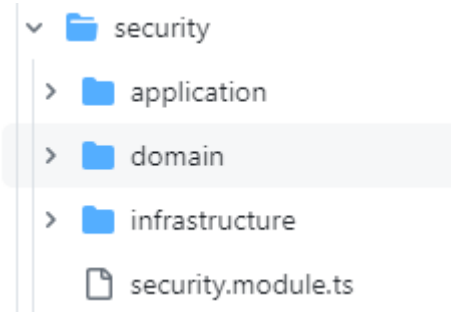
ARCHITECTURE

The architecture chosen in the assessment was “Clean Architecture” Modular, I have also worked and have experience with other architectures like Layered Architecture or Vertical Slide Architecture.

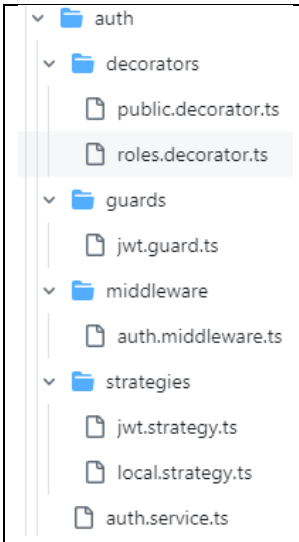
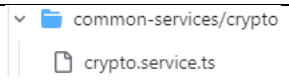
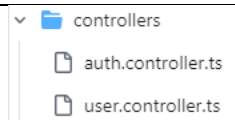
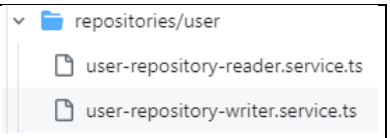
In this project the code was written in just one module (Security) for handling users, authentication, and authorization, other modules were created (empty) just to show how other features could be organized.

The Security module has three main folders that defines the “Separation of concerns” of the architecture:

- Infrastructure.
- application.
- domain.



The **infrastructure layer** contains all the services that are not part of the core but serve to the application layer, in this project it is organized in auth (authentication and authorization), common-services, controllers, repositories and validators.

 Auth handles everything relating to authentication and authorization, creating decorators to use in the controllers, guards to review the permissions, middleware to save user information that is used in audit fields (createdby, modifiedby) to save this information automatically and, strategies: local to validate user and password, and jwt to validate the token.	 Common-services will have all the services that could be used by the other modules but are related to security like Crypto Services. Right now crypto services are used to save the user password encrypted in the database.	 Two controllers are defined in this project: auth to handle the login process that will return the token and users to handle operations like creating a user, updating a user, inactivating a user, activating a user, changing the user password, and changing the user role. Each of these operations are for specific roles (normal user, staff, admin) defined by the decorator. There is also a decorator “Public” for endpoints that will not require the user to be authenticated	 Repositories are for handling the db operations, in this case I have decided to split reading and writing operations in two different interfaces and classes following SOLID principles specifically the Interface Segregation principle. I will cover later the SOLID principles applied.

The **application layer** handles all the logic that is part of the application core, like the use cases, it is organized in usescases, common-services, and interfaces (contract definition) for the infrastructure implementation.

<table>
<tr>
<td>
usecases/user
interfaces
active-user.service.ts
add-user.service.ts
change-pwd.service.ts
change-user-role.service.ts
get-user.service.ts
get-users.service.ts
inactive-user.service.ts
update-user.service.ts

The usescases implements all the business logic or use cases defined for this module, here as an example I am implementing crud operations (creating a user, updating a user, get users) and other related to possible use cases (activating a user, inactivating a user, changing user password, changing user role)
</td>
<td>
infraestructure-interfaces
auth
common-services/crypto
respositories/user
validators

Application layer does not call infraestruture services in this architecture, infrastructure services are injected in the application layer following the interfaces (contract defined) in this layer. Application services will use the interfaces to execute necessary external methods like the interaction with the database, but it is important to note that those are decoupling so it is for example in the future change to interact with other database by adding a new service and changing the injection in the dependency injection container.
</td>
<td>
common-services/audit
utils.ts

Common services contains services that are related to the security module but could be used by other modules like utils to add audit information during the insert or update db operations.
</td>
</tr>
</table>

The **domain layer** contains all the entities that are used in the application, it is organized in data-entities, service-entities, dto-entities and enums.

<table>
<tr>
<td>
data-entities
base
auditable-data-entity.ts
data-entity.data.ts
user-with-pwd.data.entity.ts

Data entities contains the entity that will be saved in the database. Here are some base classes that will be inherited for all the data entities like auditable fields
</td>
<td>
dto-entities/user
parsers
user-change-pwd.dto.entity.ts
user-change-role.dto.entity.ts
user-create.dto.entity.ts
user-login.dto.entity copy.ts
user-update.dto.entity.ts
user.dto.entity.ts

Dto entities are created for those cases that not all the data entity is used, for example request, responses
</td>
<td>
service-entities
access-token-payload.entity.ts
access-token.entity.ts

Services entities are those that are just used by services in this case for the token response and payload.
</td>
<td>
enums
roles.enum.ts
user-status.enum.ts

Enums contains all the choices for the fields that have specific options.
</td>
</tr>
</table>

(createdby, modifiedby, createdate, modifieddate)	could be different or in this case for security reason we do not want to return the password, even knowing that is encryted this should not be displayed. It also define some parsers to transform from one entity to another.		

This project represents a good starting point or template with a well-organized structure focused mainly in the architecture and SOLID principles.

SOLID PRINCIPLES

This project was focused mainly in a well organized architecture and following SOLID principles:

S – Single responsability
O – Open for extension closed for modification
L - Liskov
I – Interface segregation
D – Dependency inversion

Single responsibility. Classes created are handling just one responsibility or closely related responsibilities, for example in this project, user usecases are split into one class per usecase.

Open for extension closes for modification. It means that classes should avoid modifications but can extends or serve as base classes. By injecting services we avoid modification, if a different implementation of a service is needed we do not need to modify the existing code but just add a new class that can extend from the other one.

Liskov. A child class should be able to do everything that a parent class can do.

Interface segregation. In this project we find interface segregation in the repository, we do not have one class for all the db operations, it is split in read and write operations. We could have some cases were we just want to read information from the data base list a roles table, if we implement a interface with read and write operation we are forced to add the write operations for roles table even knowing that we will never use those operations in that entity.

Dependency inversion. As you can see this project is using a lot this principle, higher level modules should not depend on lower level modules but both should depend on abstranctions, that is why in the dependency injection container we refer interfaces and not classes, in the future if we want to change a service we do not need to modified the existing code just add the new service and change the reference in the dependency injection container.

SWAGER

In this project I have included swager that make it easier to test and document the API.