Interview Project
Invoice with different currencies

Project architecture and implementation limitations

I. Technology stack

In addition to the imposed technologies I decided to use the following libraries:

A. Database

So far I've been working mainly with Entity Framework Core but I decided to use MongoDB in this project. The main factor of this decision was the fact that NoSQL databases are perfect for storing documents.
As long as we do not need to query invoices using complex criteria against customers or billing data in general, non relational database in my opinion should be a good choice.

B. Logging

I use Serilog. It allows to log for both console and file (in opposition to default logger, which logs only to console).

C. Testing

I prefer xUnit over nUnit. I know it better and I like its functionality, specifically the ways I can load test data. I also used Moq to mock some dependency objects.

D. Docker

I don't have much experience with docker but I felt it's a good opportunity to give it a try. It's been exceptionally useful for me for working with MongoDB which I don't use a lot in every day work, so I could avoid installing mongo server on my PC and run it from docker image. It's also very convenient to deliver final app as a docker image.

II. Architecture plan

A. REST API

The requirement document specified endpoints for only one resource - invoices. In my opinion, currencies are the key resource taking important role in one of the main business processes, which is calculating invoice total amount, therefore I decided to create

endpoints also for currencies so that user could add and modify currencies which can be used in the invoices.

B. ASP.NET Core Architecture

I used standard ASP.NET Core structure. API project contains application configuration as well as request middleware and controllers. All the controller methods are asynchronous to increase performance and allow processing more requests in the same time (thread can take care of another request because it doesn't have to wait for database response). Domain project consist of domain classes representing mongo documents. I also used the repository pattern which allows me to seperate database access layer from business logic and keep each service independent from another. I used interfaces for both service and repository layer in order to provider easier implementation changes in case of changing / adding new database system.

C.

III. Known limitations

A. Security

Delivered implementation lacks many security standards which was planned due to many factors, but mainly time limitations. First of all HTTPS in this project uses self-signed certificate, which is rather mock of the certificate than a real certificate. Another problem is lack of authentication. In the real world application there should be implemented some authentication / authorization solution like for example Microsoft Identity or JWT. Identity in ASP.NET Core delivers multiple automatizations regarding authentication. It creates tables in database to store user data. There is separate controller with endpoints to register and log in a user. We use UserManager class to register new users and SignInManager to authenticate them. Once the user is authenticated, Identity automatically creates and attaches a cookie to the response for each request. It also has built in support for roles. There is an option to integrate Identity with an external service like Google or Facebook.
Alternative solution is to use JWT based security, where user authenticates outside API and sends tokens in request headers, which can be verified instead of keeping user credentials locally.

B. **Scalability**

The standard way to scale API is to use load balancer like NGINX. Alternatively Azure App Service can be used to achieve this goal.

C. **Test**

I have only written basic unit tests as an example and one pseudo integration test which starts at controller level and uses all layers down to database. It's not a real world integration test because request pipeline is not used and model validation is not triggered. To perform real integration test there should be application client used which invokes real requests to the running instance of API and receives responses, which could be tested.