I hope you are doing well. I have made the solution that was suggested, and here is my report:

Summary:
Time spent: ~50 hours
Authentication: Okta Auth0
Admin user: test@test.com pass:!@#QWE123qwe

Frontend: Blazor WebAssembly + Toast + Bootstrap + BlazoredTextEditor, as I believe a post is more than plain text. It should include every media element, and it should also be easy to render, so I picked BlazoredTextEditor as the sweetest fruit on top of the tallest tree.
Backend: C# + EF Core + MediatR and its pipelines + Unit of Work and Repository pattern + AutoMapper + Fluent validations
Database: MSSQL running on a container

Considerations:
Adding one cover image for each post is possible. Content is also saved as files corresponding to a GUID stored in the database. These images and content are not stored in the database as they should not be considered data. They are stored somewhere else.

The usage of azurite as a container that runs locally to simulate azure blob storage behaviour is presenting my understanding upon microservice architecture and how smaller modules can live in isolation

For having persistence data on the local environment there is a folder called volume being created in the root folder of the repository and it has been already excluded form the source code published on github.

Creating a reusable table component with update/delete actions was a good opportunity for me to demonstrate my problem-solving skills and deal with complex challenges. The combination of the repository pattern and functional programming paradigm applied to creating client/server contracts handled by using IHttpClientFactory strongly represents my innovation and shows my ability to design and think through the design. I also published this in a LinkedIn article, as it can be quite useful when the tech stack is the same with Blazor, MAUI, WebApi all in .net.

The use of MediatR, Repository patterns, and Unit of Work expresses my understanding and confidence in using well-known design patterns.

Coding against interfaces and being service-oriented can be easily spotted in every corner of this solution, such as retrieving userName, loadingComponent, content storage, etc., giving you a good understanding of shared spoken languages such as DRY, OOP, and SOLID.

However, since there is always room for improvement and this solution is not an exception, what can be missed here includes:
Better CSS
Validation for every operation and object
Automated testing

I decided to skip those parts as I believe the foundation is provided already, and none of them requires technical capabilities beyond what has been demonstrated in this solution. So it is just about time and the return value to take this further.

In the end, I believe it was a big thing to do as a technical assessment, but on the other hand, this means that we share the same values and we speak the same language, which makes it very interesting to work together.

Please review the code base provided in the link below, and you should be able to get it up and running by **docker-compose up --build** in the root directory if you have Docker installed on your machine.

https://github.com/siamaksh1367/FlexyBox.git
Once you clone the rope and run **docker-compose up –build**
Head to the browser and go to localhost

Please make sure to also clone submodules while cloning the repo.

Feel free to reach out to me here or on LinkedIn if you have any questions or concerns, and I am very much looking forward to hearing from you soon again