API / Server

In regards to the API part of the project I decided to keep the controller layer fairly thin and devoid of most logic outside of modelstate validation. Instead I moved my logic to a service layer placed between my controllers and my repository and database layer. I separated the concerns of the different layers by letting controllers handle viewmodel validation and responses only. The service layer handled any logic or massaging of the data and translations between domain models and viewmodels. The database / repository layer handled data access and strictly domain models. The use of view models and domain models were in order to keep the data surfacing to the controller on a strictly 'need to know' basis from the domain. Restricting the practice of sending complete domain models up to the controller layer and reducing class coupling and increasing maintainability and reusability. Because of this I moved the authorization model from the original project to a viewmodel instead as it doesn't belong to the domain in this pattern. Furthermore I added a new model called "GameListItem". This model isn't persisted but is used when I needed data from both the Ownership model and Game model to be sent to the service layer.

The repository and database layer is fairly straightforward and not very sophisticated due to the nature of the exercise and keeping data in memory. In a real world scenario they would be two seperate layers.

I used dependency injection for my services and controllers to further reduce coupling and to increase maintainability and reusability of the code.

One thing to note in the database layer is that I kept the use of your existing data access method "PrivGetData", despite it using a randomized delay when handling calls. I figured there was a reason this was added and because it even has a note saying to use it. I figured maybe it was meant to simulate latency or simply a tool to restrict the number of calls made to the data. Either way I kept it and tried to work around its limitations.

Client

On the client side I decided to use ASP.NET Core Application with React.js and Typescript. There are obviously many different ways to make the client side but I thought this was an easy and practical solution given the circumstance. I chose to use a React.js project because it's something I am familiar with and enjoy working with as well as it being a great framework.

The workflow here is a fairly easy one to get into. The “routes.tsx” file handles the routing and with the help of my static helper class “RestService.ts” ensures authentication is enforced where needed. RestService is where I keep my REST calls and helper methods in terms of authentication. In a larger project with more components and many more different kinds of REST calls I would probably keep this helper class but implement another layer of helper classes that handle the specifics of each call and how to handle the data in each specific case. As of now I let each component handle the responses on their own.

I make use of other helpers for my paths and static strings as well as localstorage access for tokens and Id’s. Beyond this it is a fairly standard react.js app.

The styling and design of the solution is rather underdeveloped and kept to a minimum. This was a conscious decision as I wanted to focus my effort on making a sound, working and scalable solution first and foremost. Design is by no means unimportant but in this scenario I simply wanted to keep it low key yet functional.

One concern with my overall solution is that I handle passwords in clear text on both the server and client side. In a production environment obviously you would invest in a SSL certificate and handle the password with hash and salt or both or some other combination of secure methods. But since this is an exercise and in development it’s simply much easier and more practical to just work with clear text.