Stellium Technical Interview - Testing

Report

Igor Kuzmin

Contents

[Introduction 3](#_Toc52721311)

[Project initialization 3](#_Toc52721312)

[Investigation steps 3](#_Toc52721313)

[Making the application work properly 3](#_Toc52721314)

[Fixing the UI 4](#_Toc52721315)

[Suggestions & recommendations 5](#_Toc52721316)

[Conclusion 6](#_Toc52721317)

# Introduction

First of all, I would like to thank you for this test. Although usually we have not a lot of time to do this kind of exercises, I think it is very useful, and this one allowed me to review some of the development concepts and learn new ones.

In the next sections you will find a kind of step by step analysis to find and fix the different issues. It may seem a bit verbose, but I have decided to do it this way to expose my way of thinking.

# Project initialization

1. Cosmos DB Emulator installation.
2. Configuration of the **Account** and **Key** variables in **appsettings.json** from the Cosmos DB Emulator.
3. In **InitializeCosmosClientInstanceAsync** method uncomment the commented TODO lines and add an item in the database.
4. Replace the line

**spa.UseProxyToSpaDevelopmentServer("http://localhost:3001");**

by

**spa.UseReactDevelopmentServer(npmScript: "start");**

to start the local development server.

1. Update Node.js and npm.
2. Now the error message appears in the browser console. To resolve it, I execute the command **npm rebuild node-sass**.
3. After that, another error:

**./src/services/APIService.ts  
Module not found: Can't resolve 'axios' in 'C:\projects\interview\_exam\InterviewExam\ClientApp\src\services'**

1. **Axios** is not installed. Install it through **npm**.
2. Now, the environment is ready. The project can be built and started.

# Investigation steps

## Making the application work properly

1. First, I am starting the application and see that there is something wrong in the CSS. But I will come back to this point later. It is just a style to adapt.
2. When I click on the Launch button, nothing happens. I am investigating the behaviour in my browser dev tools and see that the **GET** request to the **https://localhost:44331/api/item** failed.
3. The first reaction that I have is to take the bottom-up approach and investigate. I will start from the very bottom, the Data Access Layer.
4. In our case the Data Layer is Cosmos DB context and I have seen in the **InitializeCosmosClientInstanceAsync** method of the **Startup** file that there is a TODO for adding an item to the database. So, I have added it previously in the initialization section.
5. Now, before thinking about the UI, I want to check the API apart of my project. For this, I use Postman to call **https://localhost:44331/api/item** that was failed previously. I have also put a breakpoint in the **GET** method of the controller.
6. The request response was **Status 200 OK** but an HTML template was received in the response. The Controller was not called. I need to activate the controller.
7. I am going to the **Startup** class and discover that the routing is not configured properly. In the **UseEndpoints** I must map the controller routes with the next code:

**endpoints.MapControllerRoute(**

**name: "default",**

**pattern: "{controller}/{action=Index}/{id?}");**

1. Another thing that I have noted is that the request URL has not some correspondence in the Controller. From the frontend the **/api/item** was asked and, in the backend, the only existing route is **/api/item/get**. At this point I must choose one of the syntaxes and I will respect the backed syntax. Now the HTTP request is arriving properly to the controller. At the same time, I am modifying the GET method to accept the id of the item in parameters to have more meaning in the real live scenario, as we will not have only one item. Now the route **/api/item/get/{id}** is used.
2. The request is sending me the object back as intended.
3. Till now, it means that the Data and Service layers of the backend part are working properly, as well as the Web API part.
4. Let us come back to the frontend. Click to the Launch button produce the same error result in the browser console.
5. The reason is that the **APIService** call should respect the interface and I am adding the id part in the method **GetStuff(id: string)**.
6. Now I have another error in the browser console about failed **atob** function call on **Window** object.
7. After some investigations I have found that the item models from the back and front are not fully matching. In the back the name of the second property is **Description** and transform to JSON also as **description,** but the frontend is waiting for a property **content**.
8. As content name has more sense for me in this case, I am changing the back property for the JSON representation to **content**.
9. Now another error in the browser console: **URIError: URI malformed**.
10. In the **setState** method of the Home component, the received data content is firstly escaped (deprecated method) and then decoded with decodeURIComponent. We do not need the escaped method here because the decodeURIComponent can do all the job. But at the same time, I am wondering why we use decodeURIComponent. We have not encoded our content in the back so we can get rid of the decodeURIComponent too.
11. Great, the application is working now. Let us come back to the step 1 and beautify our app with the CSS. There are some problems with the positioning.

## Fixing the UI

1. Adding the next code in the .**container** class

**display: flex;**

**align-items: center;**

**justify-content: center;**

It allows me to use the CSS flexbox layout and position the content in the more flexible way.

The content is now centred horizontally and vertically as intended to be, but …

1. The logo and the launch button are on the same line. We need the logo above the button. My solution is to wrap the logo and the button in a div container. I see that currently, they are wrapped by the React Fragment with the use of short syntax **<></>**. I do not see the reason to use it, so I am just replacing it by a **<div>** tag.
2. The logo and the button are still on the same line. To fix it, I am transforming the logo to a block element with

**display: block;**

in the CSS file.

1. Another small change to do is to align the button size with the logo size. I am adding the next lines to the CSS of the button:

**width: 200px;**

**margin-top: 20px;**

and removing the next line:

**margin: 20px;**

1. That is all for the home screen.
2. By the same time, the content of the result screen is positioned correctly as well.

# Suggestions & recommendations

1. Currently, from the architectural point of view this application is hosting the front and the backend parts. I can see an improvement and leave this project as is and create a separate **.NET Core MVC** project for the API part. From this point, two possibilities exist. I can remove the controllers from the existing project and let the frontend app call the API directly, or I can let the existing controllers play the role of the API Gateway and call the true API. As usually there are pros and cons in the different solutions, depending on the size of the project, the client budget etc. We can discuss them if needed.

In this project I consider that the API part is in the same project and it was a conscious decision.

1. This improvement has a tight relationship with the first improvement. Currently **GET /api/item/get** is called to get an item. I prefer to use the RESTful CRUD naming convention **GET /api/item/123** especially for the true API project**.** But as I have considered that the API is part of the application in the previous sections, I will leave the current naming convention.
2. In the browser console we have a warning for the comparison operator that is used with only one equal sign. In **./src/components/Home/Home.tsx** let us change it to get rid of the warning.
3. Regarding the Data Layer I would create a separate data initializer class which will be called to populate the database with some data. I would also change the **Item** class name to **Image** and the container name in Cosmos DB to Image too because the Item name is too generic.
4. Another weird thing is that the content of the Item is coming from the appsettings.json file. I would move it into my data initializer class.
5. Regarding the configuration part for the web app I would also add the CosmosDb section to the **appsetting.Development.json** with only **Account** and **Key** subsections. It will allow to use the standard configuration for all the environment but Dev. But it is also a discussable subject. For example, we can just work with the only appsettings.json file and replace all the needed data during the deployment pipeline.
6. One think that I would add to the project is the Repository layer and encapsulate all the interactions with the database in it. After that, the existing Service layer will only access to that Repository layer. For example, I would create an IRepository interface and a CosmosDBRepository class that implements this repository interface. We can also create one repository by entity.
7. Even it is a small project right now, I usually create a separate Class library projects for the DAL and BLL and putting the repositories and the services there. Also, to make them fully independent I would create a model class for the Item in each of them with the automatic or manual mappings between them. But as always, I suggest taking this kind of decision together with a team. Depending on the size of the project and the potential evolution we will decide with architecture to implement. Maybe it even makes sense to move from the Monolithic design to the Microservices from the beginning.
8. Another improvement may be done with the exception management in the Service class. Currently not all methods have a try catch and some of them that have it are not doing anything in case of the exception. We should be consistent. But this improvement will not resolve the fact that currently when there is an error, our users receive a horrible exception page. So, a better solution is to move the exception handling to our controllers and only throw the business exception from the service layer. We can even go further and implement our proper middleware to handle all the exceptions and make our code cleaner by removing all the try catch constructions from our controllers.
9. One very important thing to do is to protect our API. We should use the authentication and the authorization for this. Also, different solutions exist. But right now, we even do not manage the authentication.
10. In the frontend I would split the current Home component in two components. The Home and the Content component for example.
11. In the frontend the **dangerouslySetInnerHTML** should not be used is one point. But the most shocking part is that the all the HTML content is stored and retrieved from the database. I would extract the next HTML part from the current Item:

**<img src="\*\*\*\*\*\*\*”>**

**<p>Congratulations! You've successfully launched!</p>**

And move it to the React component, leaving in the database only the image data.

1. We can also move the logo image to the database to be consistent and to have more flexibility if we want to modify our images on day. Otherwise we should replace our static images in the application project and redeploy it, or even worse replace the files directly in the web server.
2. I remember that we can use React Function Component or React Class Components. When I did a tutorial on React some time ago, I started to use the React Functions Component, but I quickly switched to the Class Component because of the state management and other advantages that I cannot remember. So, in the current code I would make that replacement. But I would like to learn more about React and discuss this one, and other points with my most experienced colleagues in this domain.
3. One more thing that we can improve is the navigation. Currently if we are already in the application screen, we cannot come back to the launch screen. It is because we have only one component and just conditionally displaying the content. As we will split the Home component in two components this task would be easily done.

# Conclusion

This is my analysis, and a list of suggestions. I have mentioned only the major points. For example, the choice of the naming conventions or usages of property with **Id** name for a string data type are not pointed in this document. Anyway, each one, and each team have their own rules and conventions. But I am sure that there are other points that we can improve that I have not discovered, for the future releases. And this is a cool part because there is still a lot of work to do.

As you can see, I have leaved some points opened to discussion and if the opportunity arises it is with pleasure that I will discuss them.

Goodbye and thank you one more time!