Mitchell Bartolo, student ID: 222086089

SIT331 Full Stack Development: Secure Backend Services

Design Document for SIT331 Activity 5.2HD

I've uploaded this project in a public GitHub repo. I'll make the repo private in a few weeks after the trimester has ended and grading is completed.

For this project I've used a variety of tech outside of what was taught during the unit. The most crucial aspects of the tech stack are:

- Node.js: A JavaScript runtime environment
- Express.js: Framework for routing with node
- TypeScript: Type safe implementation of JavaScript made by Microsoft
- MongoDB: A flexible schema NoSQL database
- Prisma ORM: A database client

I chose this tech stack because I have some experience with JavaScript, Node and Express so I generally lean in that direction. I don't have much experience with TypeScript though I had an interest in learning more.

I opted for MongoDB based purely on the desire to try something different from what we used in class. I began by following this tutorial, which introduced me to Prisma ORM and Zod for schema validation. This tutorial provided an excellent basis for getting started, and despite some issues with versions (uses an older version of Express), helped to build a solid foundation for my project. I have closely followed their suggested project structure though the majority of my endpoints and collection schema are very different.

Bounded Contexts

I have used four bounded contexts: artists, artefacts, exhibitions and users. Below is an outline of the endpoints found within each context. Each context contains the same general crud operations:

- Get all
- Get singular
- Create new
- Update
- Delete

It’s worth nothing that I’ve opted to use *patch requests* for all update operations rather than *put requests*. For the sake of the project I couldn’t think of a specific use case to include both, so I just included patch for the additional flexibility.

In addition to the standard operations, each context also contains unique endpoints, outlined below.

Artists:

- Get artists by art style: this endpoint lets users search for a specific art style and retrieves all artists who practice this art style. It validates the art style type provided by the user against the art style enum defined within the prisma schema.

Artefacts:

- Get artefact exhibitions: Returns all exhibitions that currently include or have included a specific artefact. This endpoint has been added as artefacts can be added to exhibitions, however as MongoDB is not a relational database it proved to be difficult to have this information reflected automatically in the Artefact collection. This endpoint is a workaround to make the information easily accessible.

Exhibitions:

- Get active exhibitions: Returns all exhibitions that are currently active. I went through a few iterations of accomplishing this. I wanted this to be a field that could automatically return true or false depending on whether the current date is within the start and end date of the exhibition. At first I made this a computed field (the code for this still remains in *‘src > client.ts’)*, however this solution seemed suboptimal as the endpoint was simply gathering all exhibitions then filtering based on the *isActive* computed field. Instead, I opted to filter the collection directly in the endpoint using Prismas query syntax.

Users:

- Login user: A basic login endpoint that matches the provided password with the stored hashed password.
- Update user email and update user password: Separate endpoints for updating sensitive user details.

I’ve opted not to include the schema of each context in this document. This can be seen in the file *‘prisma > schema.prisma’*. Note that the artist context was the first one created, as such I’ve included more comments in this artist controller that I haven’t repeated in the other controllers. Most of the time comments found in other controllers indicate that aspect is unique to that particular controller.

Design Decisions

Sensitive data is automatically hidden when a response contains user data. This is handled by the Prisma client, code can be found in *'src > client.ts'*. Because of this addition at the client level, it means that user details can be returned without having to worry about the sensitive data being shared, adversely when sensitive data is required (such as the login user endpoint), care needs to be taken to ensure this is not returned in the response.

Bcrypt is used to hash passwords, again this process is handled by the Prisma client so additional hashing is not required by the controller or service layer. Bcrypt is still used in the service layer however, to check the password provided by the user matches the hashed password.

I tried to decouple the Prisma ORM from the controller layer entirely, so if a new ORM was ever introduced, the resulting changes could be implemented by updating the service layer and the client only. I have been somewhat successful in this regard, though Prisma is still required in the controllers when accessing the enums defined in the schema and also for catching specific Prisma errors.

Relationships Between Contexts

One artist can have many artefacts. This was easy to implement, though this can't be observed at the database level. A one-to-many relationship defined within the schema means artefacts are automatically linked to artists when the artists ID is provided when an artefact is added.

Exhibitions can have many artefacts, artefacts can belong to many exhibitions. This is a bit hacky as the many-to-many isn't tracked how I initially expected it to be. My solution has been to create a many-to-many relationship between exhibitions and artefacts, though the validation is handled in such a way that artefacts can only be added to exhibitions, and exhibitions cannot be added to artefacts. To see which exhibitions an artefact is linked to, the *getArtefactExhibitions* endpoint was added to the artefact controller.

Further Improvements and What I'd do differently

Enhanced Errors

With extra time, the first thing I'd get stuck into is improved error messages. To begin with, these error messages have too much repeated code, I'd like to refactor to move all this repeated code into a separate utility. Secondly, improving the error messages would allow me to further decouple Prisma from the logic in the controllers, all controllers require Prisma in order to catch Prisma errors, I'd like to remove Prisma entirely from the controllers.

Testing

I was unfamiliar with most of what I was doing at the beginning of the project, and as such testing has become something of a tacked on feature at the end. With the understanding I've gained from completing the project, if I were to do it again, I would try to fully embrace test driven development. For much of the project I relied on manual testing, though it honestly became quite difficult for me to manually track all the tests I was doing as the bounded contexts are quite large. Though I did detect a number of bugs through my manual testing, I've got to assume that some bugs have slipped through.

Honestly, I'm still somewhat unfamiliar with testing, and I believe I have taken what is maybe a less complex path and opted for some simple unit tests. With more time and knowledge I'd like to implement full integration testing for more thorough testing. The unit testing I have implemented follows the singleton example found here.

Authentication

I've made a small start on authentication, though ultimately I haven't progressed to the level required. As the progress I've made is so minor, I've actually just left it commented out for the time being. The extent of my authentication can be seen in the *userController.ts* file, the *loginUser* endpoint there is code to generate a JWT token.

All in all, I've enjoyed this assignment, I've learnt a lot from completing this and I'm happy with my final submission.