Academic Reference Manager

Teacher:
Gerardo Reynaga

Teacher Assistant :
Magnús Gunnarsson

Authors/Developers:
- Daníel Ekaphan Valberg (danielv16@ru.is)
- Grétar Örn Hjartarson (gretarh17@ru.is)
- Joel Snær Garcia (joelg18@ru.is)

Context

What is the software product about?

Áslákur has provided us a list of publications, friends (users) and who has loaned what publications. The goal of the product is to make it easier to manage these information, i.e. to register and update it. On top of that reviews are also required for the publications.

What is that's being built?

To solve this we are building a RESTful web API that helps with the management of this data. Our job is to implement all the routes and functionalities Steven Tímóteusson has provided us. However, we are not using the same technologies he has designed the system but we are working on a similar structure. This is a NodeJS application (JavaScript runtime environment see https://nodejs.org/en/), using express (see https://expressjs.com/) as the web framework

How does it fit in the existing environment?

Because there are no existing service specifically made for this purpose in this environment, our product will be very suitable for the users and the owner of the publication library.

Who is using it?

Outsiders who are interested in the collection and would like to know what publications and their rating will get to explore it. Then there are users who have more access than outsiders will be able to loan and review publications. Then there is the owner, who can do everything other users can, as well as manage (e.g. delete and create new publications into the database).

Functional Overview

What the system does

The system supports CRUD (Create, Read, Update, Delete) operations for everything, i.e. publications, users, loans and reviews. There are three types of users the system needs to support, with separate levels of functionality:

1. *Anonymous users*, which can only view the publication library and the reviews
2. *Authenticated users*, that can:
    a. Borrow and return publications
    b. Give a review of a given publication (and modify the review)
    c. Get recommendation on what publication to borrow next
    d. Get a report of all the publications which are on loan on given date
    e. Get a report of the reviews given by users
3. *Administrators* have full access and can modify anything. Furthermore the administrators can:
    a. Add new publications and users
    b. Output a report of users and publications that have publications on loan on a given date
    c. Output a report of publications and users that are on loan on a given date
    d. Output a report of users that have had a publication on loan for a given duration, e.g. longer than 30 days

The user type is currently being passed through the “Authorization” header and being checked with a middleware in each route that just checks user types.

Quality Attributes

Performance

With our current system and data we get results in under 1 second (no exact time as connection to database can be varied). So far there has not been any performance hits. We have not done any capacity testing as of writing this, but as time goes on and database expands we will have to do so, and evaluate our code where needed. We are also currently using non-relational database with relational data which could be heavily impacted, so migration to relational database might have to be done in the future (after evaluation).

Monitoring and uptimes

So far we have not implemented any monitoring system and yet to deploy to real world application to be able to see uptimes, however in most cases we are handling service side errors and server should be running in those cases. Logs are also yet to be implemented to log errors on automated restarts. Client side errors (such as data not found) are also handled by the server.

Constraints

Our main HR constraints for this project is our time. All group members had a busy schedule up until the last 4-5 days of the deadline. This impacted us mostly on our ability to write thorough tests for everything, however we managed to do integration tests and Data Layer (queries and such) tests which were what we felt were the most important parts to test. Doing integration test in a sense also tested our service layer as edge cases were tested, knowing the data layer fetches correct data and that our routes were getting expected results was sufficient to know if our service layer worked correctly.

Other than that were just technical constraints. For example, of technology stack, but we were fast to learn and tackle problems. As well as deployment knowledge (although we are not required to do so in this project), but would be good to do proper stress/capacity tests on deployment.

Principles

We mainly focused on the following principles:

- We are using a 3-tier architecture layering structure throughout. I.e. we have a Controller layer (presentation layer), which handles all the routing and calls the corresponding service functions and sends correct responses to the user. Then we have the Application or Service layer, which handles logic for the application. Finally a Data layer, which communicates with our database and queries the data required for the service layer.

- Our data layer does not compose of explicit logic, but logic are used to query the right data for our application usage, as we believe that to give a better performance than letting the service do data binding after multiple queries as each queries can add significant time. These were and should be tested well.

- DRY (do not repeat yourself), we broke the code down into modular parts as well as we could and reused a large portion of the code where we could.

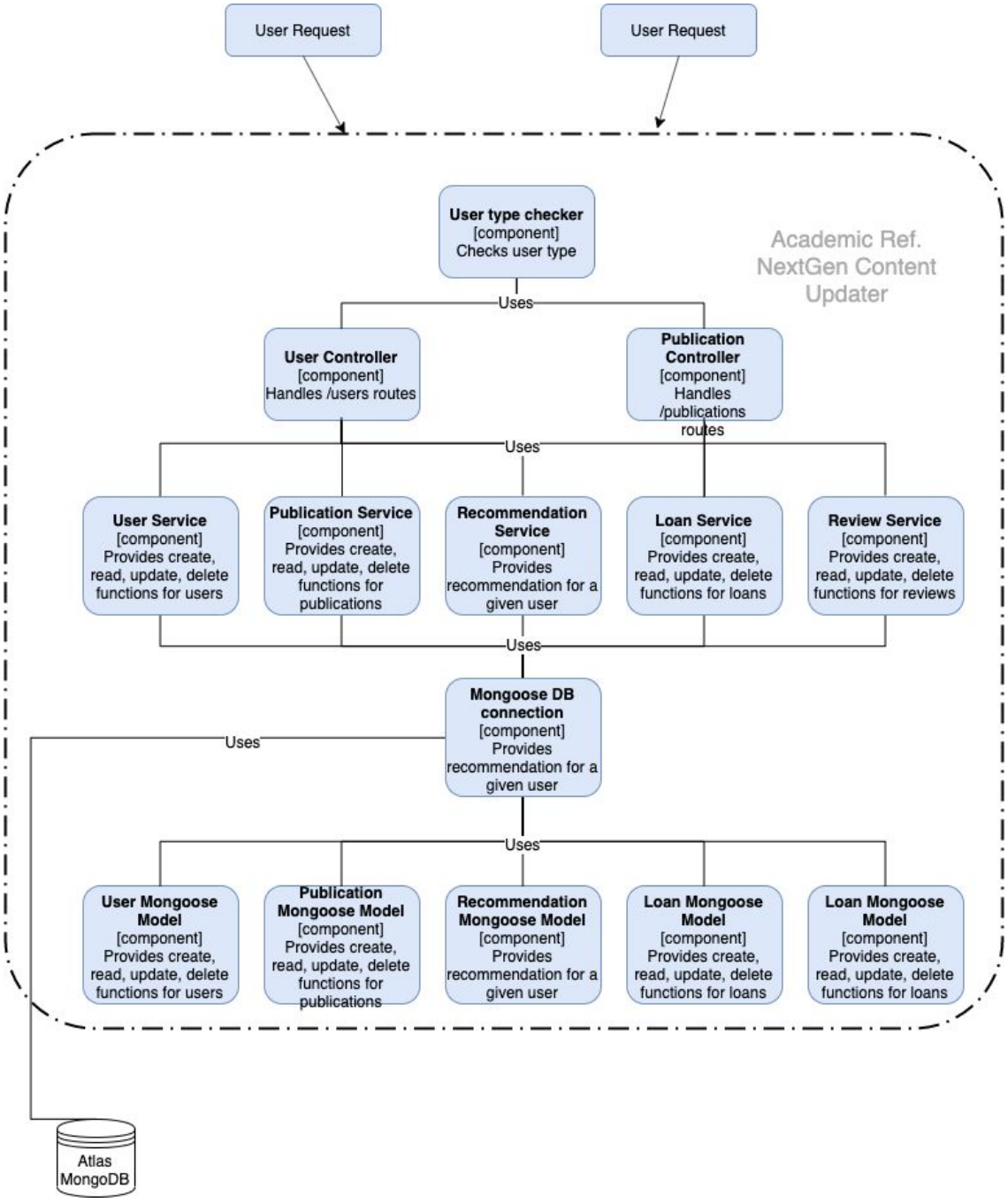
- IoC (Inversion of Control) and DI (Dependency Injection) was heavily used throughout the project. Our IoC component was holding onto all the dependencies we required and were injected using the IoC context. Thus our code is mostly loosely coupled, and are easily unit testable. Apart from some components that did not really need dependency injection. Files are then initialized with factory pattern (using context).

- We try to keep our naming schemes readable, descriptive and as similar as we can throughout all the layers of the application to make it easier to navigate through the code and know what to expect. However parameter names are not as strict.

Software Architecture

Component Diagram

Our components are based on similar structure to the one given to us by Steven Tímóteusson with slight modifications, here is the updated version.



External Interfaces

We are using external NPM packages in our application. These can be found in the *package.json* file under “dependencies” and “devDependencies”, these have good reputation and are well documented by the third-parties and are quick to fix issues as many developers depend on them.

Currently we are using the following:

- Dependencies
    - "@hapi/boom": "^8.0.1",
        - For creating and throwing http response codes
    - "express": "^4.17.1"
        - Web framework for NodeJS
    - "moment": "^2.24.0"
        - Handle time
    - "mongoose": "^5.7.7"
        - Mongodb object modeling and connection
- devDependencies (just used for development)
    - "jest": "^24.9.0"
        - Testing framework
    - "nodemon": "^1.19.4"
        - Watch file changes and restart server on change while developing
    - "supertest": "^4.0.2"
        - HTTP assertions, used with jest.

Code

As mentioned, we are using 3-tier architecture to structure our codes, so we carefully follow the rules.

Since this is a NodeJS application there are no strict typing (would be otherwise if written in TypeScript, but none of the members has experience on TypeScript). We are not using any of the *Class* ES6+ syntaxes but just functions and prototypes. We chose not to use any callbacks where we can and use ASYNC/AWAIT syntaxes to handle asynchronous codes (avoiding callback hells!).

All the source code is resided in */src* folder, everything outside are not relevant to the main application. However there is the */initData* which contains the original data (json files) from Áslákur and a script to parse to the right format with relations and seed it into databases. The collection it sends the data to depends on the *NODE_ENV* environment variable.

For dependency injection we are using the IoC folder, *context.js* contains all the dependencies and *context.js* to inject dependencies. Files do not directly import any dependencies, only the context file and imports on initialization (factory).

The 3-tier architectures and split into the following folders:

- */routes* - Controllers that handles routing from user requests
- */services* - Service (application) layer that does the heavy lifting
- */db/* - data layer where Mongo Schema (*/db/schemas*) and queries (*/db/statics*).

Each layer may not communicate in parallel and cannot communicate upwards (e.g. services cannot call controllers), but a higher layer may communicate with multiple lower level components.

Again we try to be descriptive with method names rather than use abstract naming.

Data

We are using a third party hosting provider: https://www.mongodb.com/cloud/atlas

As mentioned before we are using Mongoose to define our schemas, queries and establish connection to our mongodb server. These can be found under */src/db*. The connection key and authentication can be found in */src/config*.

We initialize the data from Áslákur into the database. The database is split into 3 collections, i.e. test, development and production and are used in respective environments.

Infrastructure Architecture

We yet do not have experience in deploying our server to proper production environment, however we have setup a script to run in production mode (seen in README.md).

Deployment

All installation and deployment instructions can be found in REAME.md