Dumitrescu Vlad-Eduard
Group: 258-1

Bookshop App - Documentation

Project Description

The proposed project represents a bookshop application, in which users may perform various operations over a database of books.

From the Home page, the signed-in users can see the image, title, and price of each book stored in the database. For each book, they can see more details, like the author, old price, or its rating. Every user has the possibility of adding a new book to the list, by filling in the above-mentioned info. Also, he/she can edit or delete his own books (those that were added by him/her). Moreover, they can make a discount on their books, which means that the current price will become the old price, while the actual price will be decreased by $5. All users will be notified when such a discount occurs. Last, but not least, users can see their own books from the MyBooks page and all discounts within the bookshop, from the Discounts page. The authentication and authorization are implemented with the help of Auth0. The non-logged users are only able to see the Home page.

System description

The system, which is implemented using NestJS for the backend, and React for the frontend part, consists of several technologies, including microservices, micro frontends, third-party integrations, and containers for deploying the solution.

The User-Interface (UI) part contains two micro frontends, named *"bookshop-app"* and *"view-book-app"*. The first one has the main core of the application, while the second one embodies the *"Book"* component, used when clicking on the *"view"* button. Actually, this app has its module exposed and is incorporated within the former, by using the Webpack Module Federation bundler.

These micro frontends communicate with the backend part through an API gateway, which is a microservice. This service receives requests from the clients (micro frontends) and it sends them to the corresponding microservice, according to a specific command, with those microservices executing the required operations. In order to do that, the service uses messaging patterns. Hence, the web server exposes REST services through the gateway. Also,

the gateway sends server-side notifications to the UI, by using web sockets. Whenever a discount for a book is made, this is seen by users as notifications on the screen.

All the logic is handled by the other two microservices to whom the API Gateway communicates. One of them, *"bookshop-server"*, contains the CRUD methods over the Book object: finding all books, finding a specific book, creating a new book, updating or deleting a book. The other service, "discounts-server", contains three methods, for performing a discount on a book (update the price and the old price of a specific book), getting the logged user's books (return the books which were added by him/her) and getting the discounts within the bookshop (return the books that have the price less than the old price, or those that are free (i.e., they cost $0)). The messaging between each microservice and the API gateway is done with the aid of RabbitMQ, which is a message broker supporting multiple messaging protocols.

The database that is used by these two microservices contains the *Book* model and it's implemented in MongoDB, a no-SQL database. The connection between it and the services is realized thanks to Mongoose.

The system is integrated with the Auth0 third-party service, which represents an authentication and authorization platform. The paths for creating, updating, or deleting a book, but also all paths corresponding to the *"discounts-server"* service (update the price, get the discounts, get my books) are secured with Auth0, under the *"api-gateway"* service, in the *"App.module"* file. So, we can safely affirm that the REST services are secured.

Finally, the last part worth discussing is represented by the Docker service, used to deploy the system, by packaging the applications as portable container images. For each service (the two micro frontends and the three microservices, including the API gateway), a "*Dockerfile"*, having all the necessary commands for building an image, is created. All these files are then wrapped up inside the *"docker-compose.yml"*, which basically starts all containers together (together with the MongoDB database).

To resume, these are the concepts and technologies used, for each part of the system:

1. **Web server:**
    - has microservices implemented in NestJS, using message patterns and RabbitMQ as messaging protocol
    - has an API gateway that exposes the secured rest services by using Auth0 and links the micro frontends with the microservices
2. **Web app:**

- has micro frontends implemented in React; one frontend uses the other, which is exposed, by using the Webpack Module Federation plugin
- receives server-side notifications from the gateway, when a discount occurs, with the help of web sockets

3. **Integration**
    - third-party integration with Auth0, securing the REST services
4. **Containers**
    - Docker containers for each service
    - docker-compose to deploy the solution (to start all the containers)

Software Oriented Architecture Patterns

Some of the patterns used by the system are:

- Microservice pattern: the backend is designed as a collection of loosely coupled services
- Micro frontend pattern: the frontend monolith is decomposed into smaller parts that can be developed and deployed independently
- Messaging pattern: there is asynchronous messaging for communication between the gateway and the services
- API gateway pattern: a service that represents a single entry point to all clients
- Service instance per container pattern: each service is deployed in its container

Diagrams

We present here three UML diagrams, corresponding to the three microservices, a c4 system diagram and a c4 container diagram.

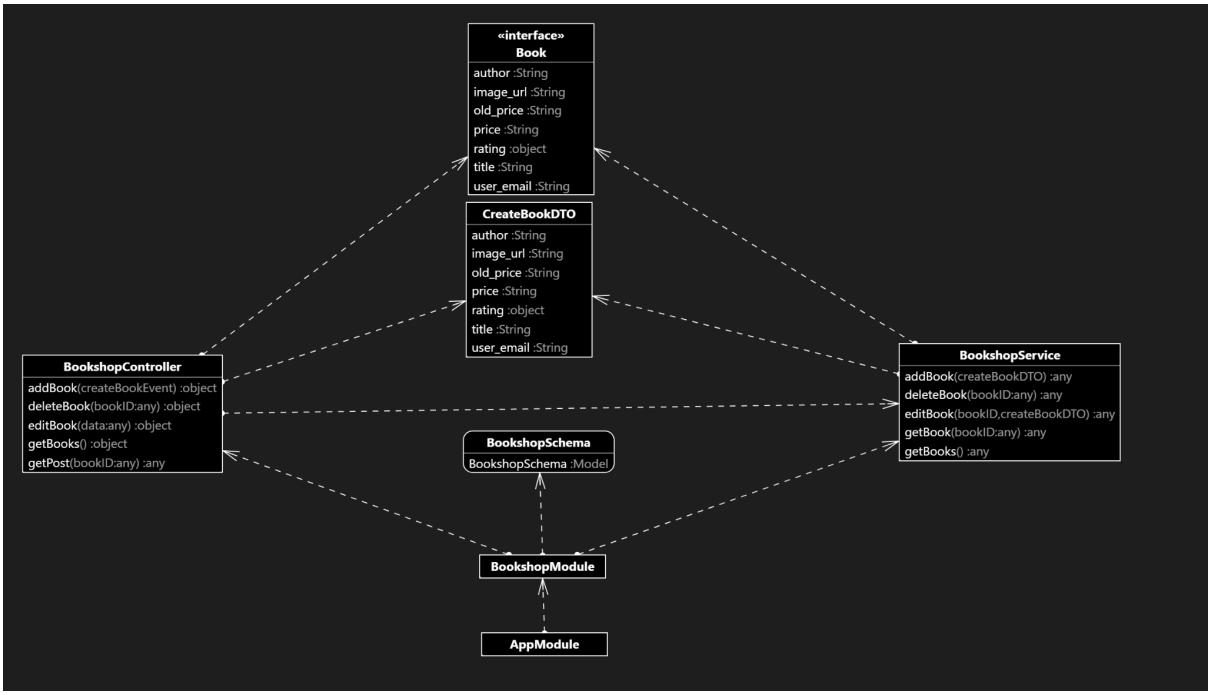

Fig. 1. UML diagram for bookshop-server

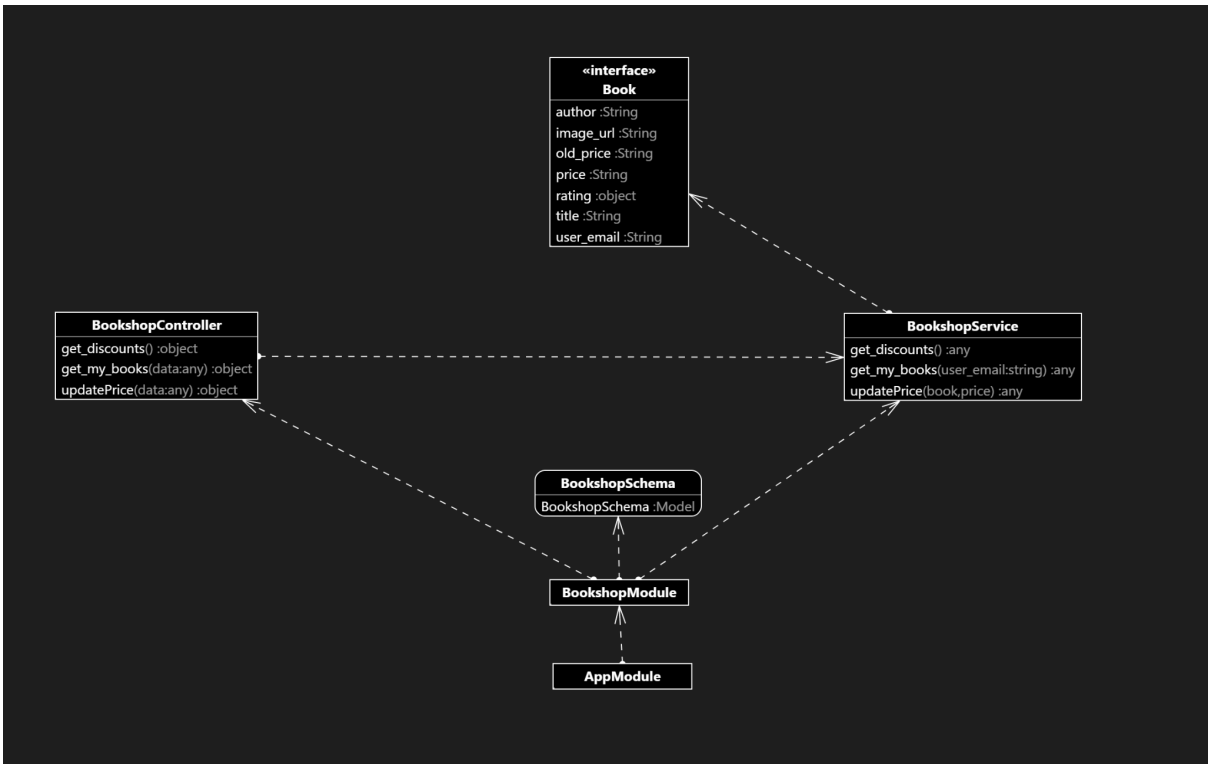

Fig. 2. UML diagram for discounts-server

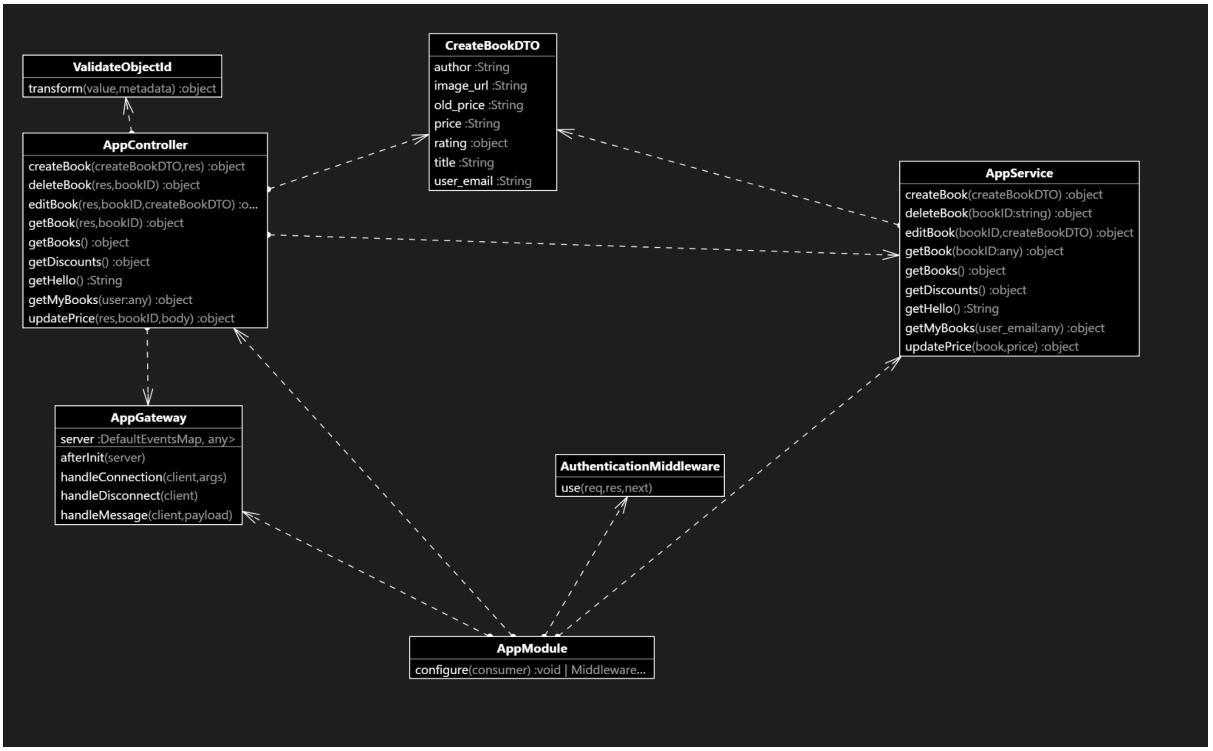

Fig.3. UML diagram for api-gateway

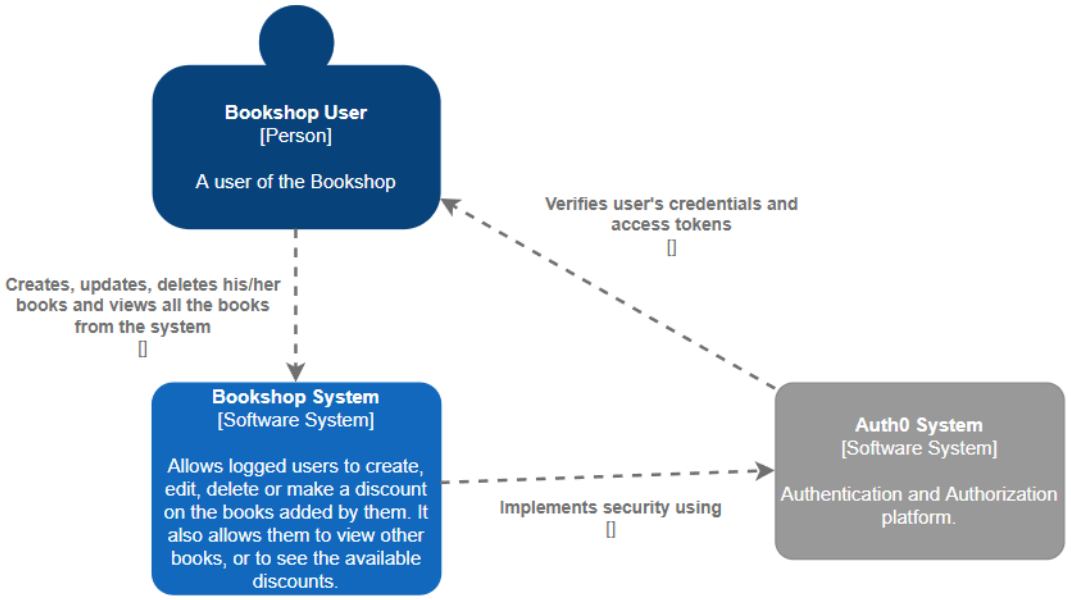

Fig. 4. c4 system diagram

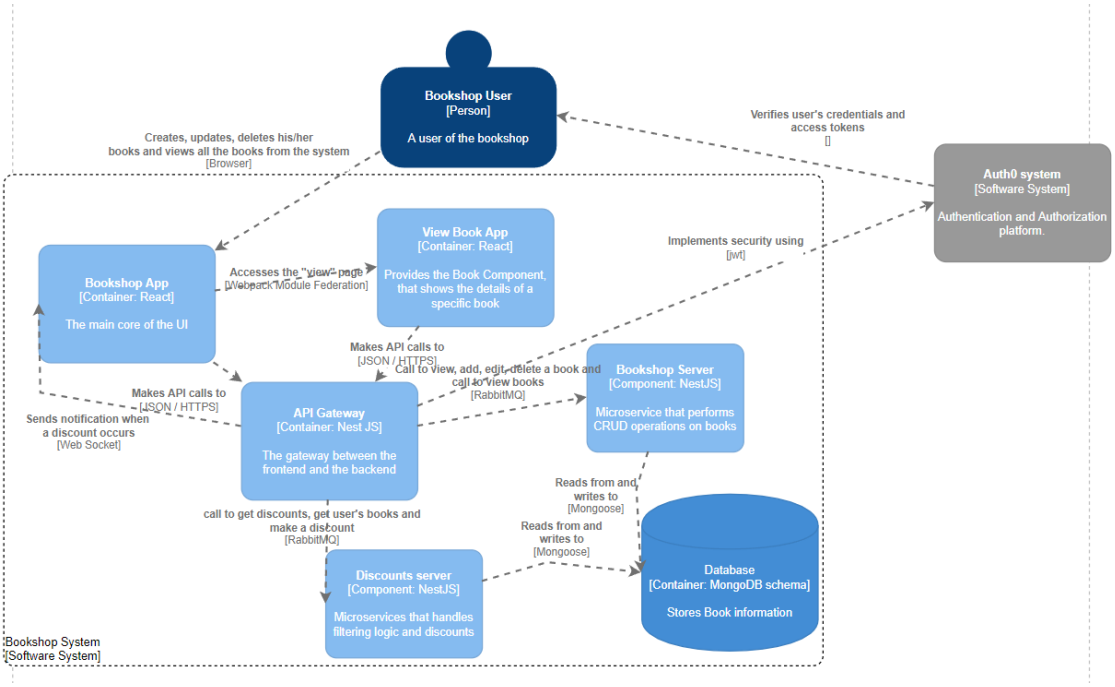

Fig. 5. c4 container diagram

Installation steps

- inside the project, from the command line, run `docker-compose up`

- run the app by accessing http://localhost:3000 in any browser.