PRODUCT REVIEW APP

Requirements and System overview

Hugo Miranda (1201669)
Štěpán Zelenka (1230017)

Index

1. System overview 2
   1.1. Initial system overview 2
   1.2. Current System Overview 3
2. Architectural Significant Requirements (ASRs) 3
3. Architectural Issues 3
4. Proposed Architectural Enhancements Rationale for Design Choices 4
5. Initial system vs. System-as-is 5
6. Issues 8
   6.1. Recommendation algorithm data 8
   6.2. SKU Generator interface 8

1. System overview

1.1. Initial system overview

The initial system was designed as a Maven-based Spring Boot application, functioning primarily as a Product Review platform. The core functionalities can be seen in Figure 1 The application was limited to using a Java Persistence API (JPA) database and did not support multiple database types, lacking extensibility and configurability in this regard.

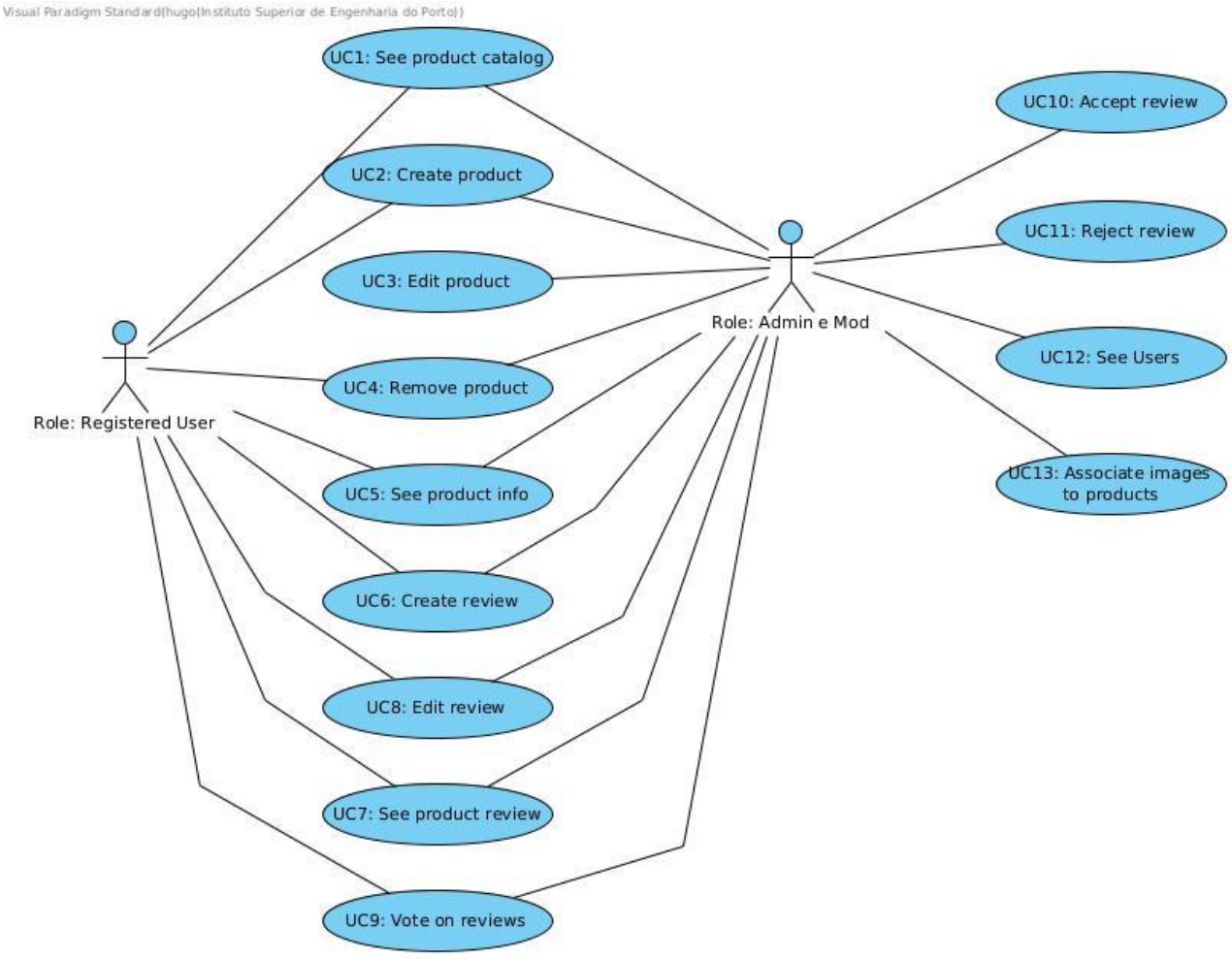

Figure 1 – Use case diagram

1.2. Current System Overview

The system has significantly evolved with the introduction of two new SKU generators and two review recommendation systems, all configurable via *application.properties*. The system now supports MongoDB and Neo4j databases in addition to JPA, enhancing its versatility and adaptability. The models and repositories have been expanded to cater to each database type, ensuring that the service layer remains untouched for increased modularity.

2. Architectural Significant Requirements (ASRs)

The ASRs were focused on enhancing the system's flexibility to handle various database types through configuration settings. Additionally, the introduction of SKU generators and review recommendation methods needed to comply with the Open/Closed principle for extensibility and configurability, ensuring the application adheres to best practices and maintains the onion architecture. We went ahead and specified some custom names for each SKU generator and reviews recommendation algorithm.

For the SKU generators the requirements were:

- 1st SKU - “Random Pattern”: alternate a digit with a letter 3 times -> alternate a letter with a digit 2 times -> add 2* special characters that are neither a digit nor a letter.
- 2nd SKU - “Hash based”: product name -> SHA-256 -> hexadecimal -> 12* middle values

* We added 1 more character on the 1st SKU and 2 values on the 2nd SKU generators to have a normalized SKU of size 12.

The review recommendation service requirements were:

- 1st review recommendation - “Popular”: For a review to be recommended, it must have 4 votes and at least 65 % of them must be upvotes.
- 2nd review recommendation - “Vote based”: If the user votes at least fifty percent as me (this means matching upvotes and downvotes), all reviews of the user (that I have not yet voted for) will be recommended to me.

3. Architectural Issues

The main limitation encountered in the initial system was its inability to handle multiple database types, which was a crucial aspect of the provided ASRs. Moreover, the system's lack of configurability meant that it could not adapt to changing requirements or extend functionalities without significant rework.

4. Proposed Architectural Enhancements Rationale for Design Choices

To overcome these issues, the system's architecture was enhanced to be driven by configurations defined in *application.properties*. This involved creating separate models and repository implementations for each supported database type and allowing these to be interchangeable based on the configuration, without altering the service layer.

This method of extending the application's capabilities without modifying its existing behavior is commonly referred to as a "strategy pattern". In our context the "strategies" are the different database types (JPA, MongoDB, Neo4j) and the different SKU generators and review recommendation systems that can be configured in the *application.properties*.

For the repository's we used the “Repository Pattern” we created interfaces for the repositories, and then implemented these interfaces for each type of database.

This approach adheres to the Open/Closed Principle—one of the SOLID principles of object-oriented design—which states that software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.

We used the “repository pattern” because of the abstraction of the data layer, providing a separation of concerns. It allows the application's service layer to remain agnostic and it also helps decouple the business logic from data access logic, which simplifies maintenance and evolution of the system. Basically, it aligns perfectly with our requirements. Other options would include using direct data access that means no abstraction and would tightly couple the business logic to the data access technology.

We made a choice to use the “strategy pattern” as one of our requirements was to be able to have the flexibility to switch between different algorithms or strategies at runtime based on configuration. Other options would be the use of if-else statements but that would make the app impossible to maintain.

5. Initial system vs. System-as-is

On the Figure 2 we see the diagram that intercepts the package and logical diagram that illustrates the high-level design of the Product Review Application's backend (BE) system on the initial project.

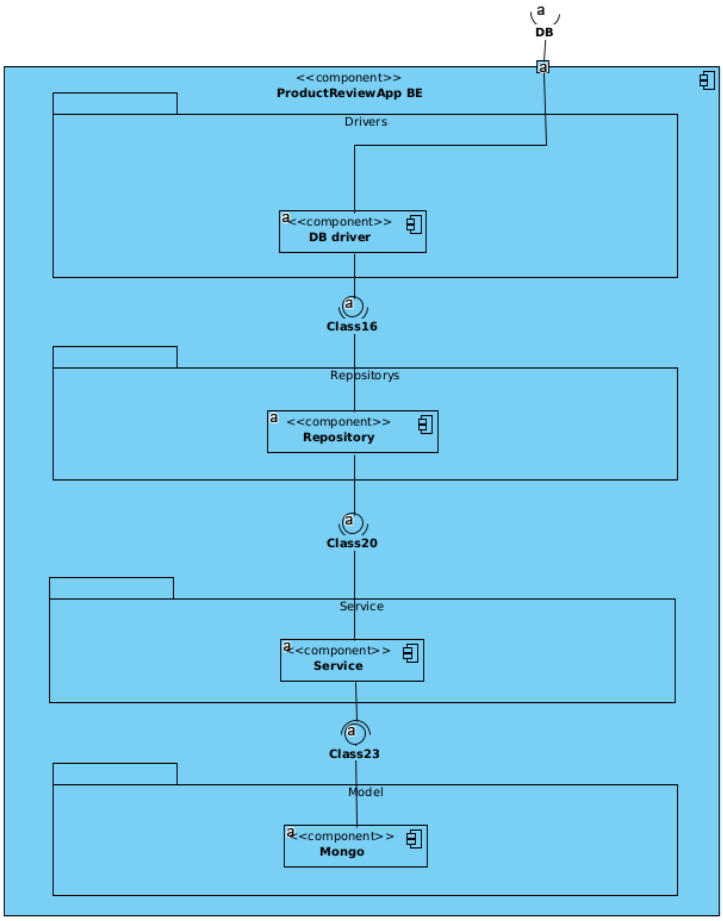

Figure 2 – initial system package logic diagram

If we compare it with the current diagram of Figure 3, we can see the architectural evolution from the initial to the updated system showcases a shift from a single-database design to a modular, multi-database approach. The first diagram represents a basic model with JPA database integration. The second diagram reveals a refined structure, supporting JPA, MongoDB, and Neo4j, with clear segregation of components for each database type. This enhances flexibility, allowing for easy extension and configuration, and a direct response to our requirements.

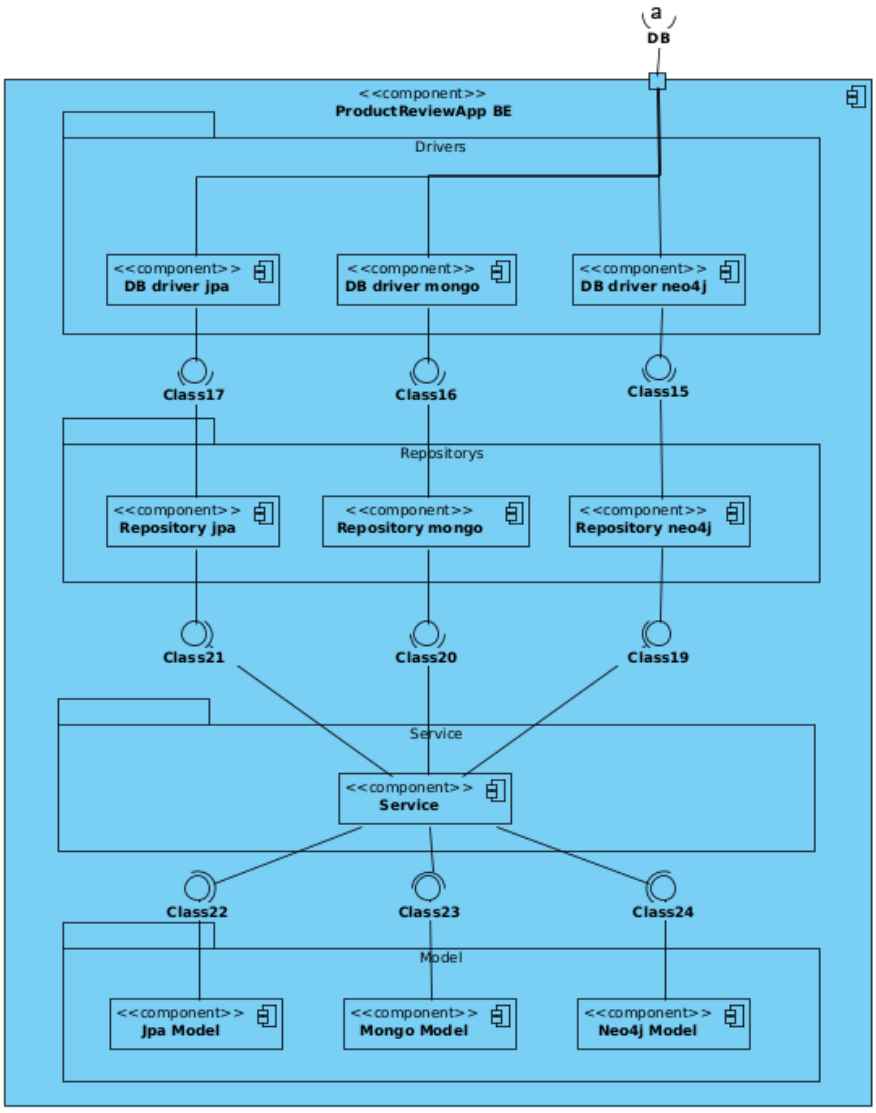

Figure 3 – system-as-is package logic diagram

Overall, these architectural enhancements align with the principles of open/closed design and offer a robust foundation for future growth and evolution of the application.

Example of use of the *aplication.properties* configurations:

```
## Recommendation Algorithm (VoteBased, Popular)
app.recommendation-algorithm=VoteBased

## SKU Generator (HashBased, RandomPattern)
app.sku-generator=HashBased

# values: Relational, Graph, Document
app.database.type=Graph
```

Figure 4 – application.properties configurations

6. Issues

6.1. Recommendation algorithm data

With implementing the recommendation algorithm based on popularity, an issue (or a decision) arose. We can either get all the reviews from the repository and then filter the reviews based on the specified conditions. Or we can filter the reviews already in the database (which can be done in all the implementations).

The pro of the first approach is that we follow the onion architecture concepts, separating the layers in a way that the repository does not know (and does not care) about the business logic of the recommendation algorithm. It just provides data. The con is that this approach may get slow, depending on the number of reviews in the database.

If we were to choose the second approach, the infrastructure layer with the repository would then contain the business logic, which belongs to the domain layer. We can improve this by naming the data fetching method of the *ReviewRepository* interface to something like 'getReviewsWithAtLeastFourUpvotesAnd65PercentOfUpvotes' (or put this info into a *JavaDoc*), but this is still only a name and does not enforce anything; the repository implementation can just go ahead and implement whatever. Then, the best way to enforce it would be to write an integration test with an actual database.

6.2. SKU Generator interface

The random implementation does not need the product's name to work, while the hash-based implementation does. But both need to implement the SKU Generator interface – this is the business requirement. How would this be solved?

Our thoughts are a clarification of requirements with the client and then changing the interface/implementations.