CS-319

Deliverable 5

Team 06

Group Members

Muhammad Rowaha - 22101023

Maher Athar Ilyas - 22001298

Mehshid Atiq - 22101335

Ghulam Ahmed - 22101001

Ismail Özgenç - 22001648

Date: 03/12/2023

1. Class Diagram

1.1 Class Diagram Key

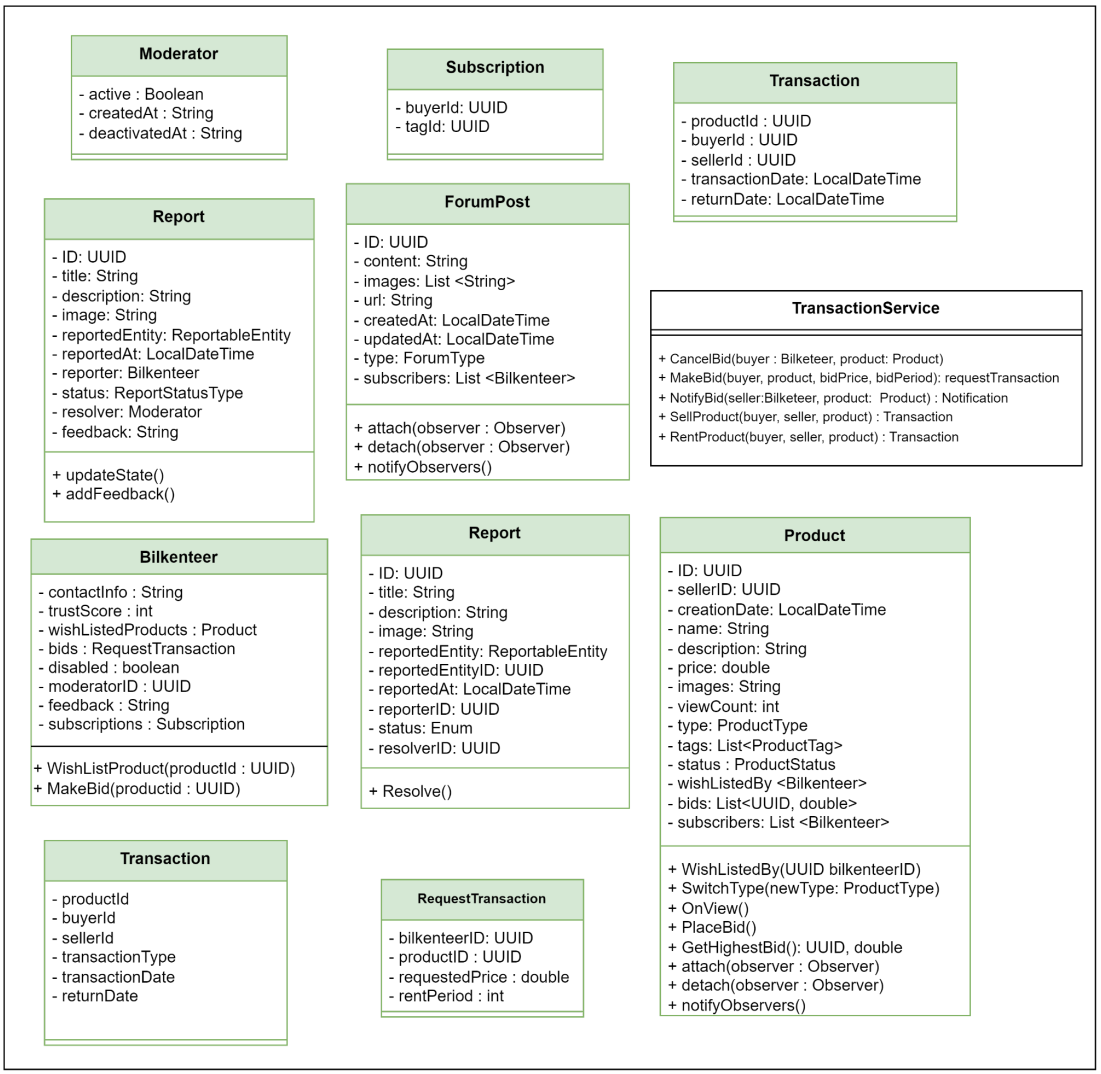
Yellow Classes: UI Classes
Blue Classes: Controller Classes
Green Classes: Entity Classes
Purple Classes: Repository Classes that extend JPARepository in Java

Note:

1- All Entity Classes have getters and setters for all fields added by Lombok notation.

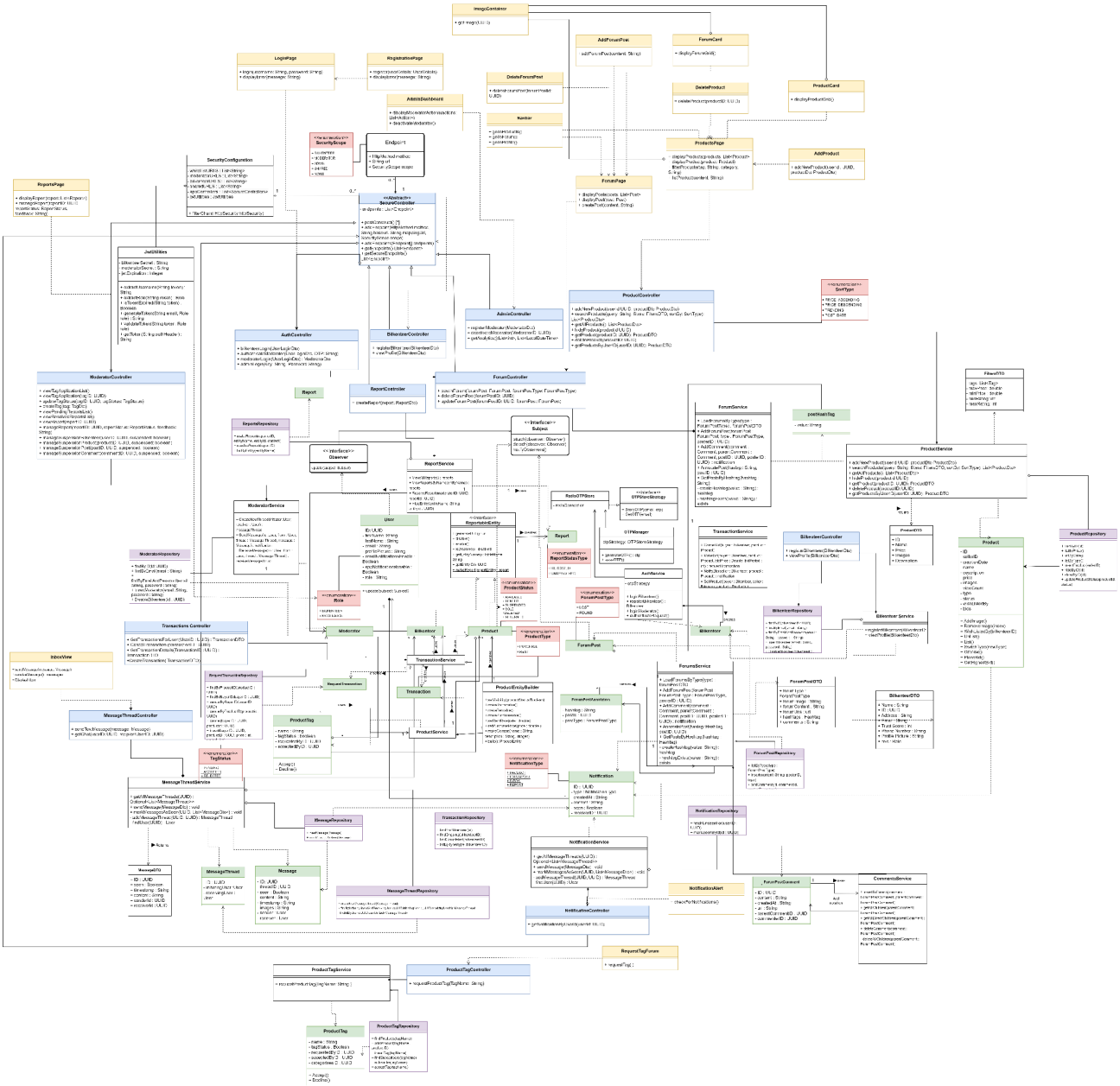
1.2 Expanded Classes

Some classes are minimized in the class diagram to support readability. Their expanded version is below.



1.3 Class Diagram

For higher resolution: https://drive.google.com/file/d/1JUJBYC8UR4_qZHQRCovgQPSa8katDdos/view?usp=sharing



2. Design Patterns

2.1 Builder Pattern

The Builder pattern is useful in our context as it provides a flexible solution for constructing the complex Product Entity, which can have various states and types, such as being purchasable, rentable, borrowable, listed, hidden, sold, rented, or disabled. To ensure consistency and integrity of the Product instantiation process, the class employs a private constructor, making it inaccessible from outside the class scope. This encapsulation forces the creation of Product instances to go through the ProductEntityBuilder, which offers a controlled and step-by-step construction process. This builder class ensures that a Product object can only be created with a valid state and all necessary attributes set, preventing incomplete or inconsistent states that could arise from direct construction. By delegating the object creation responsibility solely to the ProductEntityBuilder, the design also enhances readability and maintainability, as the construction logic is centralized in one place, making it easier to manage and evolve the product instantiation logic over time.

2.2 Facade Pattern

The classes in the repository layer provide an abstraction
over the raw SQL queries that will be performed on one or more related entity
classes the repository may be dependent on. By extending the JPA repository and adding query methods within the repository classes, we allow this layer to serve as a facade that abstracts the complexities of direct database manipulation and encapsulates the complex SQL operations. For example, methods like findTransaction(buyerID, productID), will use the BilkenteerEntity and ProductEntity to perform SQL queries and joins before returning the Transaction list. Moreover, the implementation of all service classes is done to provide intermediate APIs to other services so that the complexity of developing individual services is reduced. This not only makes the repository layer more accessible for consumers but also promotes loose coupling and separation of concerns, as service classes interact with these repositories through clear, simplified APIs.

2.3 Observer Pattern

We have used the observer design pattern to implement the notifications related to Products and ForumPosts. Here, the User implements the Observer interface, and Product and ForumPosts implements the Subject interface, providing appropriate means of subscribing and unsubscribing to the subject entity. The subject entities maintain a subscribers list that is used to notify all subscribers about any update on the subject entity. This design pattern not only simplifies the notification logic but also ensures the timeliness of information sent to users. It offers a scalable solution where new subscribers can be easily integrated into existing subscriber lists without disrupting the notification flow for existing subscribers.

2.4 DTO Pattern

For nearly all interactions with the user interface, we utilize Data Transfer Object (DTO) abstractions instead of directly communicating with complete entity objects. The rationale behind this approach is that not every attribute of an entity is required for a specific UI action. For instance, when returning a message, it is unnecessary to transmit the full User Entity for both the sender and receiver. Instead, we employ the MessageDto, which includes senderId and receiverId, rather than sender and receiver objects as in the Message Entity. This method significantly reduces data transfer, thereby saving bandwidth and enhancing processing speed. Furthermore, it plays a crucial role in security by concealing attributes that are not required for the user's action, ensuring sensitive information is not exposed. The use of DTOs here is helpful in creating efficient, secure, and user-centric applications.