BILBOARD

Team-T4 BugBunny

Class Diagram and Design Patterns

DILARA MANDIRACI

BURAK DEMIREL

YUSUF TORAMAN

SILA ÖZEL

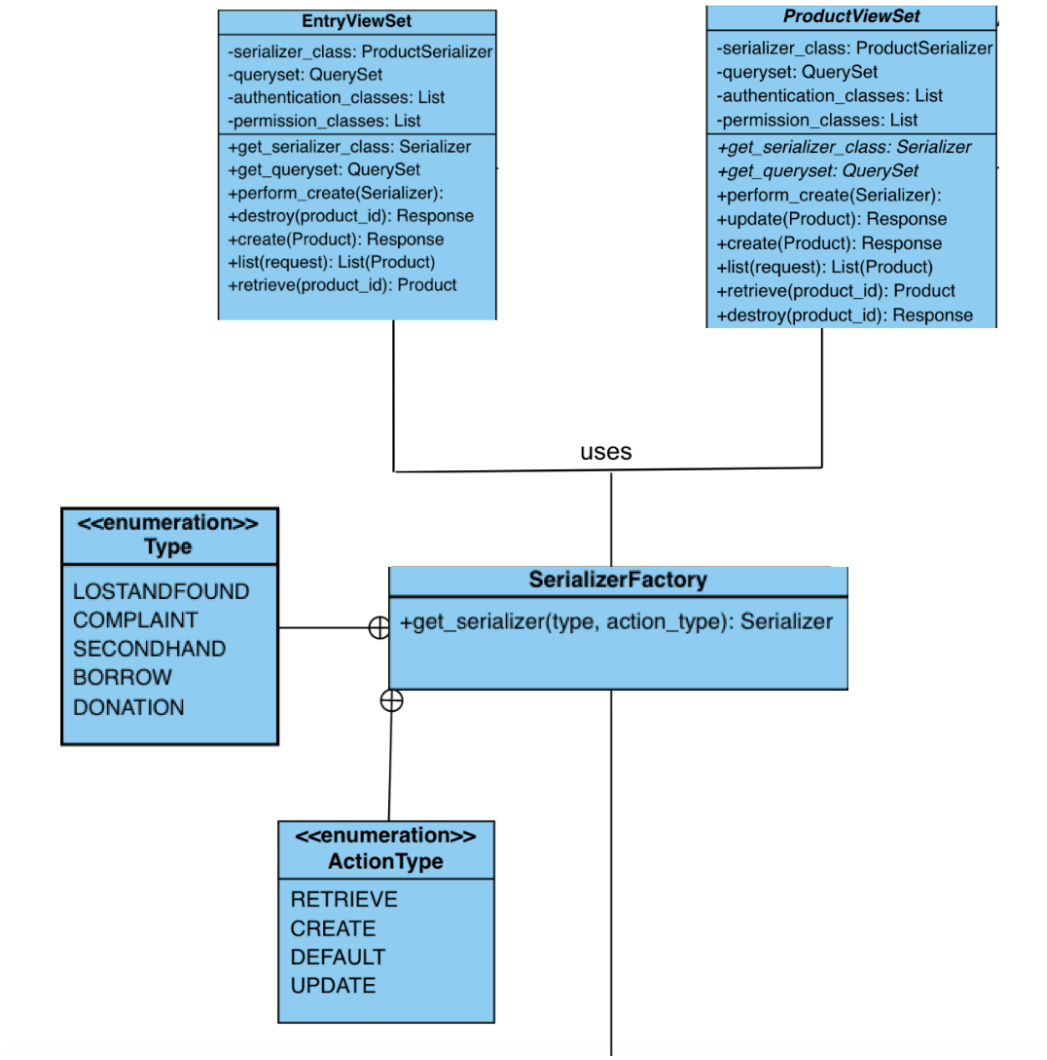
EREN HAYRETTIN ARIM

Fall 2023

Table of Contents

1. Design Patterns

1.1. Factory Pattern



General

In our Django REST Framework project, we implemented the factory pattern to manage the serialization process efficiently across different modules like Complaint and Lost and Found. This approach is encapsulated within the SerializerFactory class, which stands for creating serializer classes.

Enum Classes

Firstly, we introduced enumeration classes, namely Type and ActionType. These enums represent various types (e.g., LOSTANDFOUND, COMPLAINT, SECONDHAND) and action types (e.g., CREATE, RETRIEVE, UPDATE, DEFAULT). This categorization improves the clearance and maintainability of our codebase, making it easier to add or modify product types, entry types, or action types in the future.

SerializerFactory Class

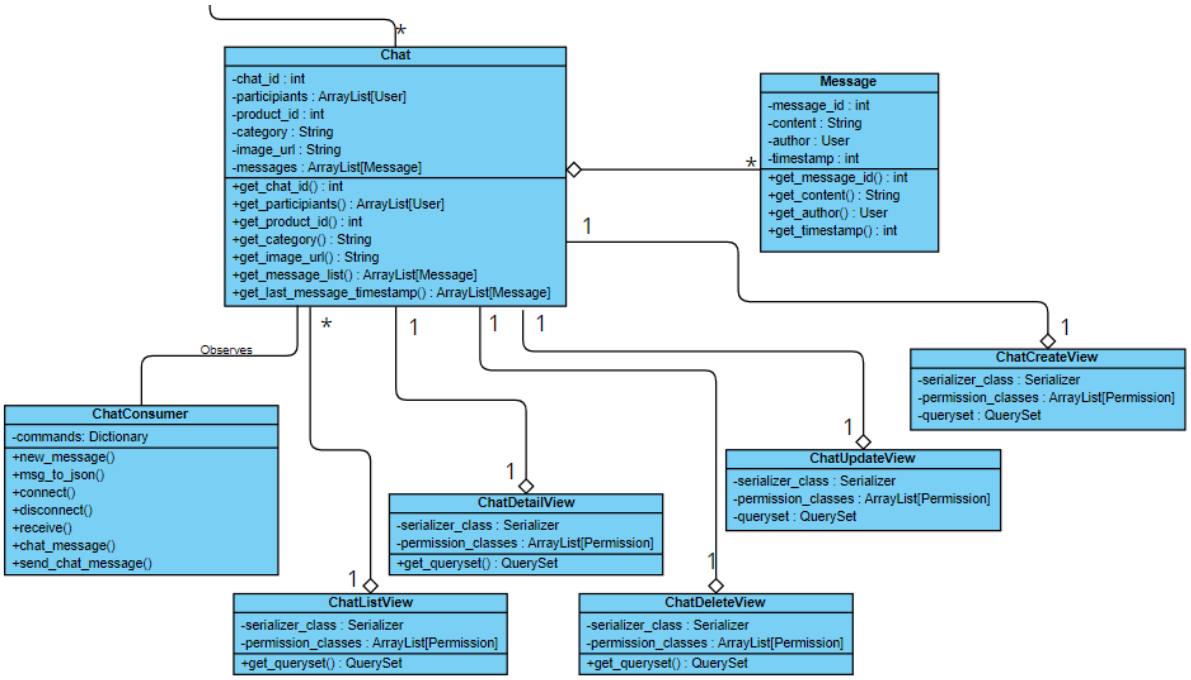
The SerializerFactory class is our core class. It contains a method get_serializer, which takes Type and Action as parameters. Depending on these parameters, the method returns the appropriate serializer class. For instance, for a LOSTANDFOUND Type with a CREATE Action, the CreateLostAndFoundEntrySerializer is returned. This dynamic selection mechanism allows us to adapt to varying requirements without hardcoding the serializer classes in our views.

Usage in ViewSets

In viewsets, such as UserLostAndFoundEntryViewSet, LostAndFoundEntryViewSet, and UserComplaintEntryViewSet, we override the get_serializer_class method. Here, the SerializerFactory class is used to determine the serializer class based on the current context. Moreover, this pattern is also valid for the products. However, it is just mentioned for Entries to demonstrate the logic and the pattern.

Factory method pattern not only makes our viewsets extensible but also reduces code duplication, especially when similar types of serializers are used across different viewsets. Also, the standardized serializer creation makes our code more tidy and increases its readability.

1.2. Observer Pattern



In our Django REST Framework project, we used the Observer Pattern to implement real-time messaging. This approach is encapsulated within the ChatConsumer and Chat classes.

Chat Class

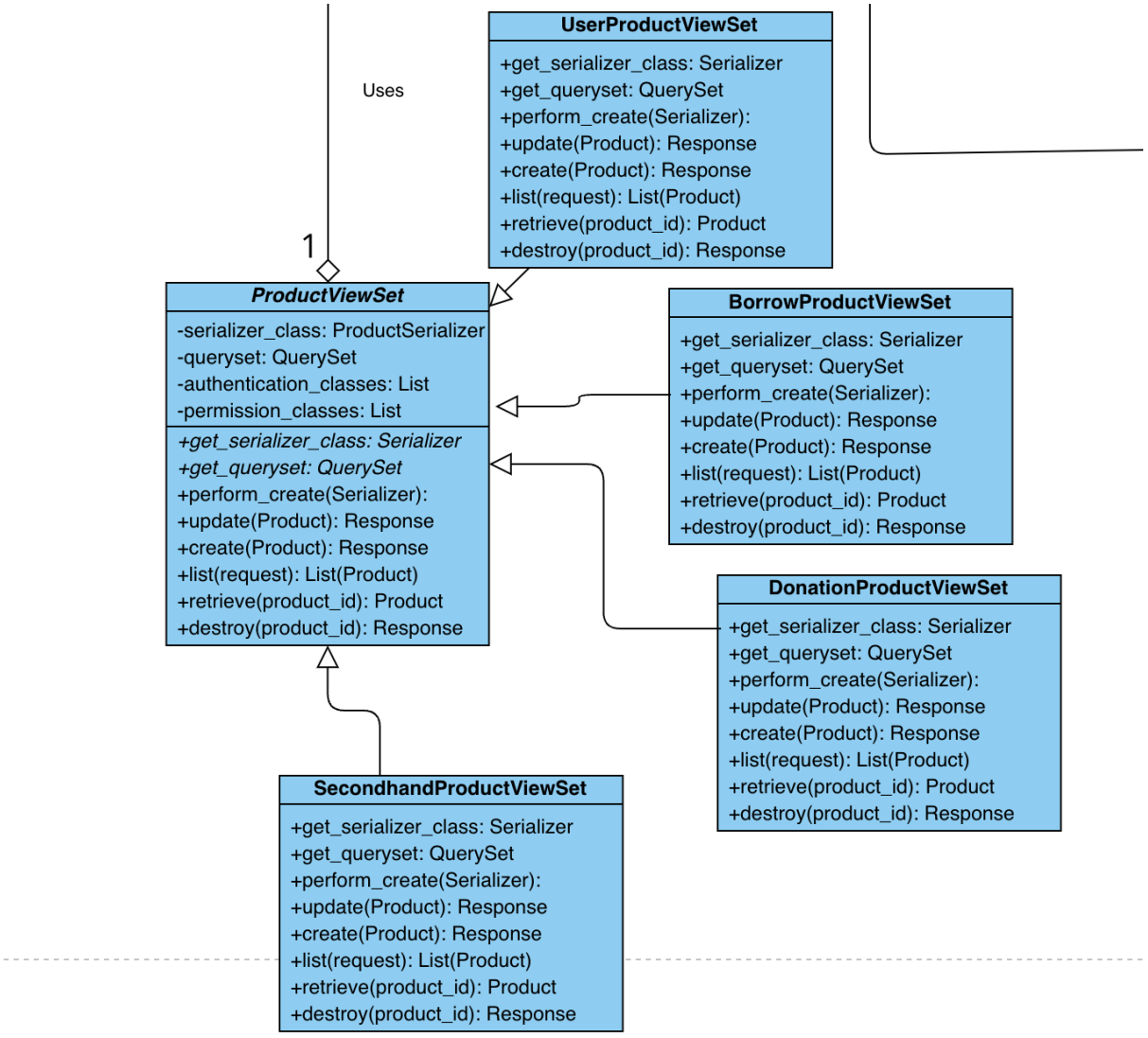
Chat class is dedicated to the subject of the observer pattern. It is the actual chat that holds the participants, messages, and product-related fields.

ChatConsumer Class

Changes related to the Chat class are observed via the ChatConsumer class with a WebSocket. When a message is sent via WebSocket, this request is caught by ChatConsumer. ChatConsumer recognizes the change in the Chat class and saves the message to the database. Afterward, it notifies WebSocket back so both users can see the newly texted message on their screen.

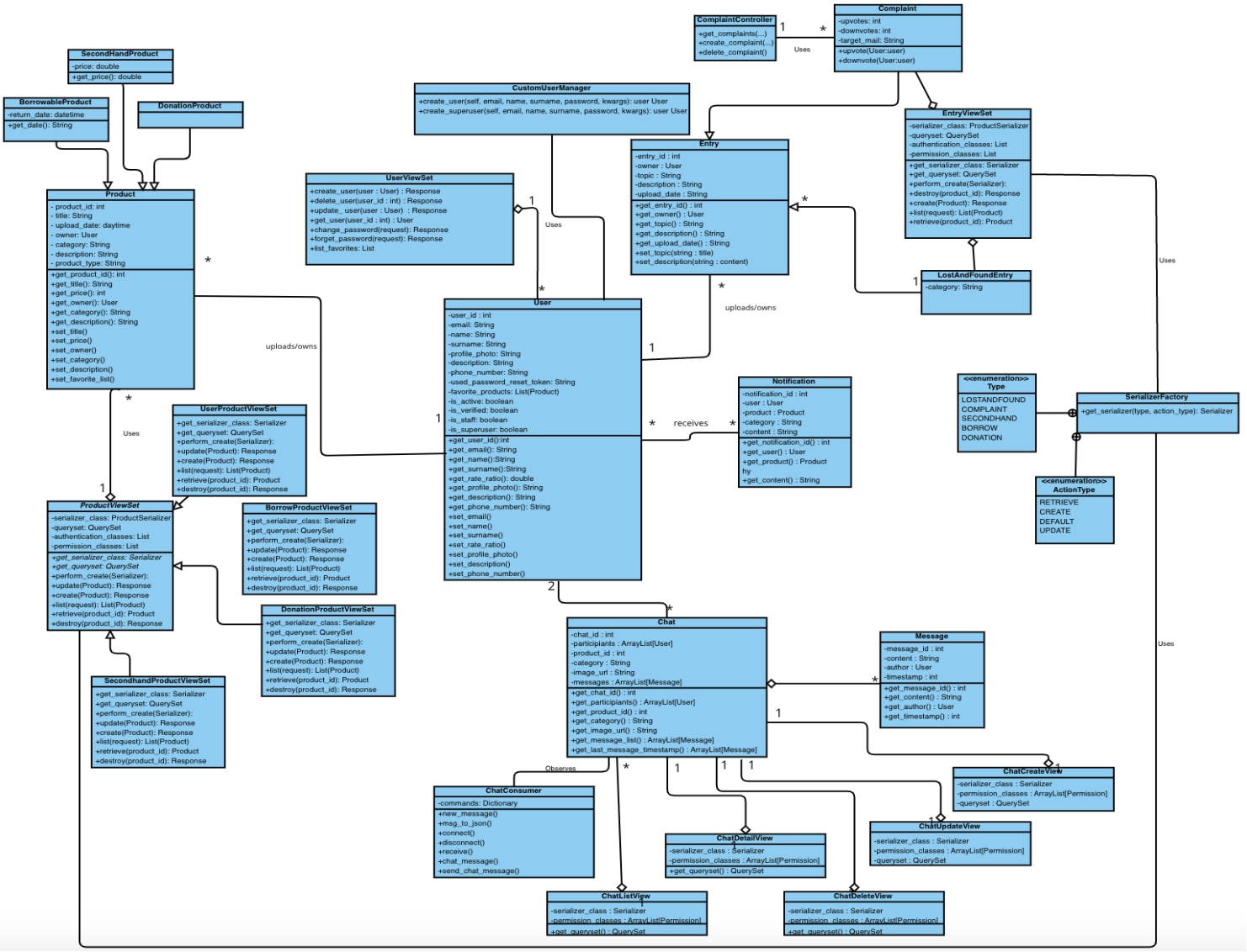
The observer pattern makes Chat and ChatConsumer quite independent. So, any other consumer class can be defined to observe the Chat class without needing to modify the fields or methods of the Chat class.

1.3. Template Pattern



In our application, for Product operations, we have implemented an Abstract Base Class named ProductViewSet. All product classes (secondhand, donation, borrow) inherit from this base class, and abstract functions are overridden according to their product types. We used this pattern because SecondhandProductViewSet, DonationProductViewSet, BorrowProductViewSet, and UserProductViewSet share common functionalities, but these functionalities must be implemented differently. In order to achieve that, we decided to use the template pattern.

2. Revised Class Diagram



Changes about User

- The UserViewSet now has additional methods, indicating more sophisticated user interactions like forget_password, list_favorites, etc.
- The User object holds favorite_products as a list with a ManyToMany field. It means that one user can have many favorite products while one product can have many users who sign it as a favorite product.

- used_password_reset_token attribute is added to User. Via this attribute, our app controls the reset token of the User. It checks whether the User tries to use this token twice.
- CustomUserManager class, which has a relationship with User, has been added. The fields are set correctly when creating the user, and the password is hashed first to ensure privacy and security in this class.

Changes about Product

- New classes for handling product views like UserProductViewSet and SecondhandProductViewSet are added. These classes are the child classes of ProductViewSet, and they perform every operation related to the products. For example, UserProductView handles the requests of the authenticated user. SecondhandProductViewSet handles the secondhand product operations like creating a new secondhand product and deleting a secondhand product.
- get_favorite_list operation is removed from the Product class because now it is handled in the UserViewSet.

Changes about Chat

- The introduction of the Message class implies the implementation of a single message sent to the database by a user.
- The introduction of the Chat class implies the implementation of a chat between two users.
- The ChatConsumer class provides real-time chat capabilities with WebSockets on the frontend.
- ChatListView, ChatDetailView, ChatCreateView, ChatUpdateView, and ChatDeleteView classes are added to manage API endpoints of the chat application.

Changes about Entry

- In our previous design, LostAndFound and Complaint objects were independent models. However, this design was changed, and both Complaint and LostAndFound objects are now child classes of the Entry class.
- LostAndFound and Complaint models have been modified to use EntryViewSet methods.

- Category was added as an attribute to the LostAndFound model to separate Lost uploads and Found uploads. In this way, when a listing is made, the Lost advertisement and the Found advertisement can be distinguished from each other.

Relationship Changes

- ### User Interactions
    - The relationships between user classes and product classes have been detailed, indicating specific interactions such as views, ownership, and actions.
- ### Messaging System
    - The relationships between Message, ChatConsumer, and user-related classes are designed to provide a complex messaging system involving multiple users and groups.

Changes in Serializer structures

- SerializerFactory class added to the productapp and entryapp (these are apps that we have created in Django for modularity of our code). Thanks to this class, serializers are returned more regularly. In addition, two enumeration classes named Type and ActionType were added to be given as parameters to SerializerFactory. The SerializerFactory class is associated with both EntryViewSet and ProductViewSet and returns the necessary serializers for the necessary operations.