BearGo

Iteration 2

(Project Vision, Requirements, Sequence diagrams, design model, business layer, deployment diagram, component diagram, architecture, demo with CRUD REST API, identification of GRASP patterns, OCL, the revised analysis)

AGM Islam
Grad Student, Computer Science
agm_islam1@baylor.edu
Baylor University

Maisha Binte Rashid
Grad Student, Computer Science
maisha_rashid1@baylor.edu
Baylor University

Razwan Ahmed Tanvir
Grad Student, Computer Science
razwan_tanvir1@baylor.edu
Baylor University

Swapnil Saha
Grad Student, Computer Science
swapnil_saha1@baylor.edu
Baylor University

Tonni Das
Grad Student, Computer Science
tonni_jui1@baylor.edu
Baylor University

Nov 18, 2022

Contents

1 Project vision

The vision of the project is to create a platform for people to send items to their known acquaintances in a convenient way.

Functional Requirements

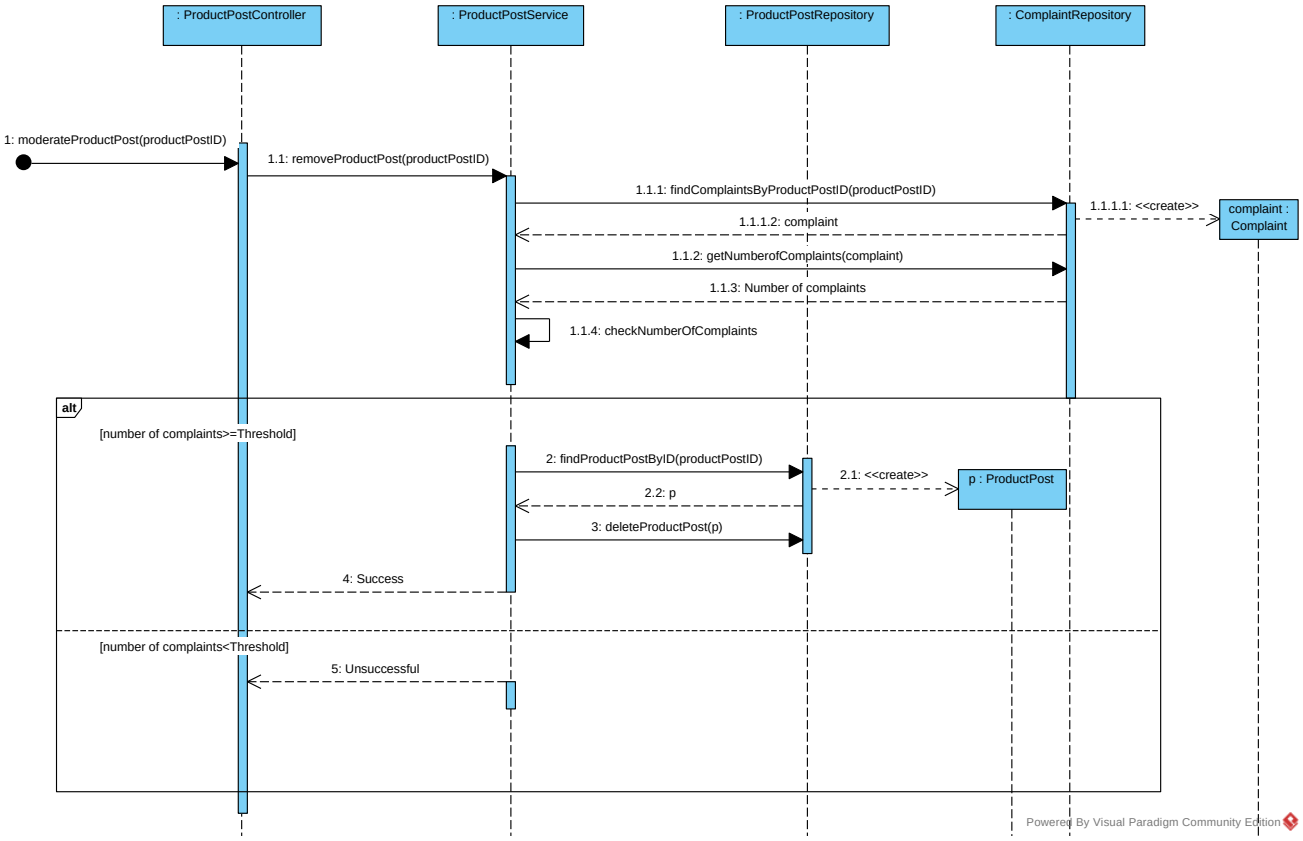
1. Product Tracking: Traveler will update the status of different stages of the product delivery. The sender will finally confirm if the product is delivered.
2. Contracts of User: Admin will be able to see any user's activity such as what deals s/he has made so far and what are the status of those deals.
3. Report Review Confirming: Admin can choose to personally talk to reporters of a ProductPost or just notify them that the post was removed.
4. Create ProductPost: Product information posting (routing info, pick up location & travel time) in application feed to find a traveler who can deliver the product.
5. Comment of ProductPost: Comment section under sender's ProductPost in which users can write comments if they have any queries regarding the productPost or other relevant details.
6. Create BlogPost: Registered users can upload their travel stories as blogs in Travel Blogging option.
7. Search ProductPost: Users can search other senders' productPosts based on different filters such as - date, location.
8. Ban User: User can report against another user and admin is responsible to ban a user from our platform.
9. Update ProductPost: Senders can also update the productPost. Senders can change the date, route, and pickup location. But once an invoice has been made or the delivery has passed then sender can not update the productPost anymore.
10. Contract Creation: Sender will create a contract and after agreeing with a suitable traveler, he will confirm the contract. The system will generate an invoice.
11. Moderate ProductPost: Admin will decide to remove/keep the productPost if the productPost is reported by the users.
12. Send Notification: Real-time web push notification and email notification. Notification will be triggered on different events like: agreement signed, products delivered.
13. Review and Rating: When the contract of a productPost will expire, both traveler and sender will see an option there to post a review and rating(1 to 5) to each other.
14. Peer-to-peer Chat: Any traveler/sender can contact to any particular sender/traveler to discuss the product details or any other travel information.
15. Social Media Share: A sender can share his/her productPost on social media.

2 Sequence Diagram

2.1 AGM Islam

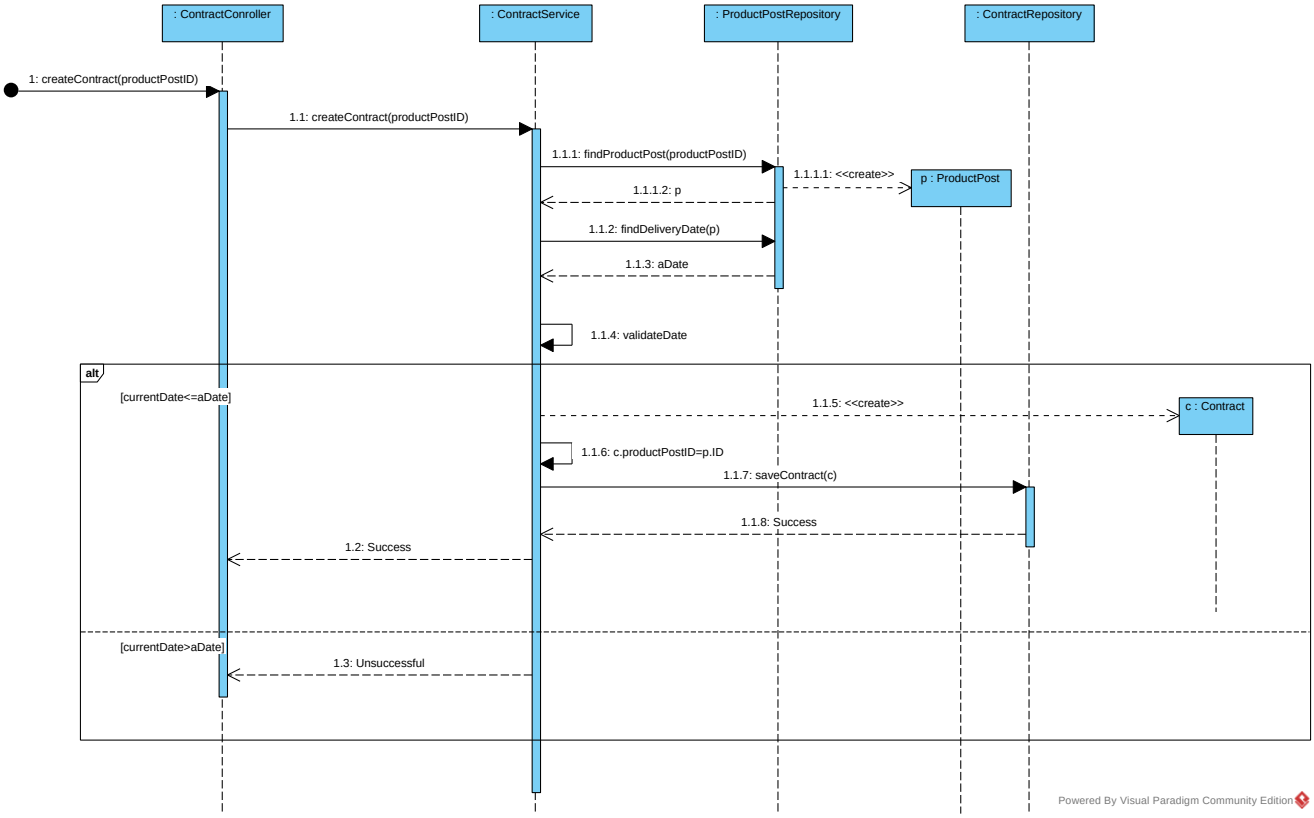
2.1.1 Operation Contract: moderateProductPost.

Use Case: Functional Requirement 11: Moderate ProductPost.
Operation: moderateProductPost(productPostId)



2.1.2 Operation Contract: createContract.

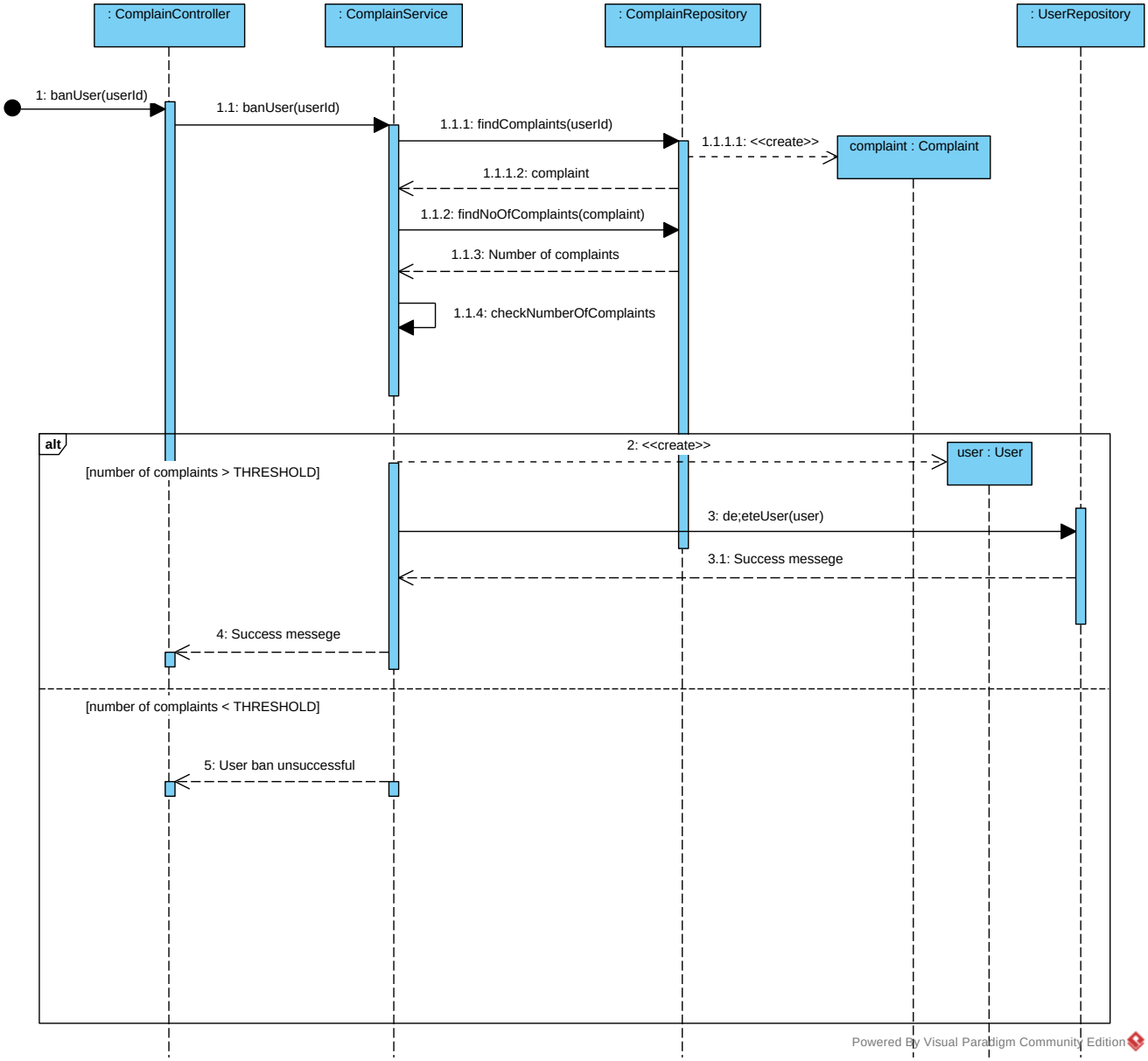
Use Case: Functional Requirement 11: Create Contract.
Operation: createContract(productPostId)



2.2 Maisha binte Rashid

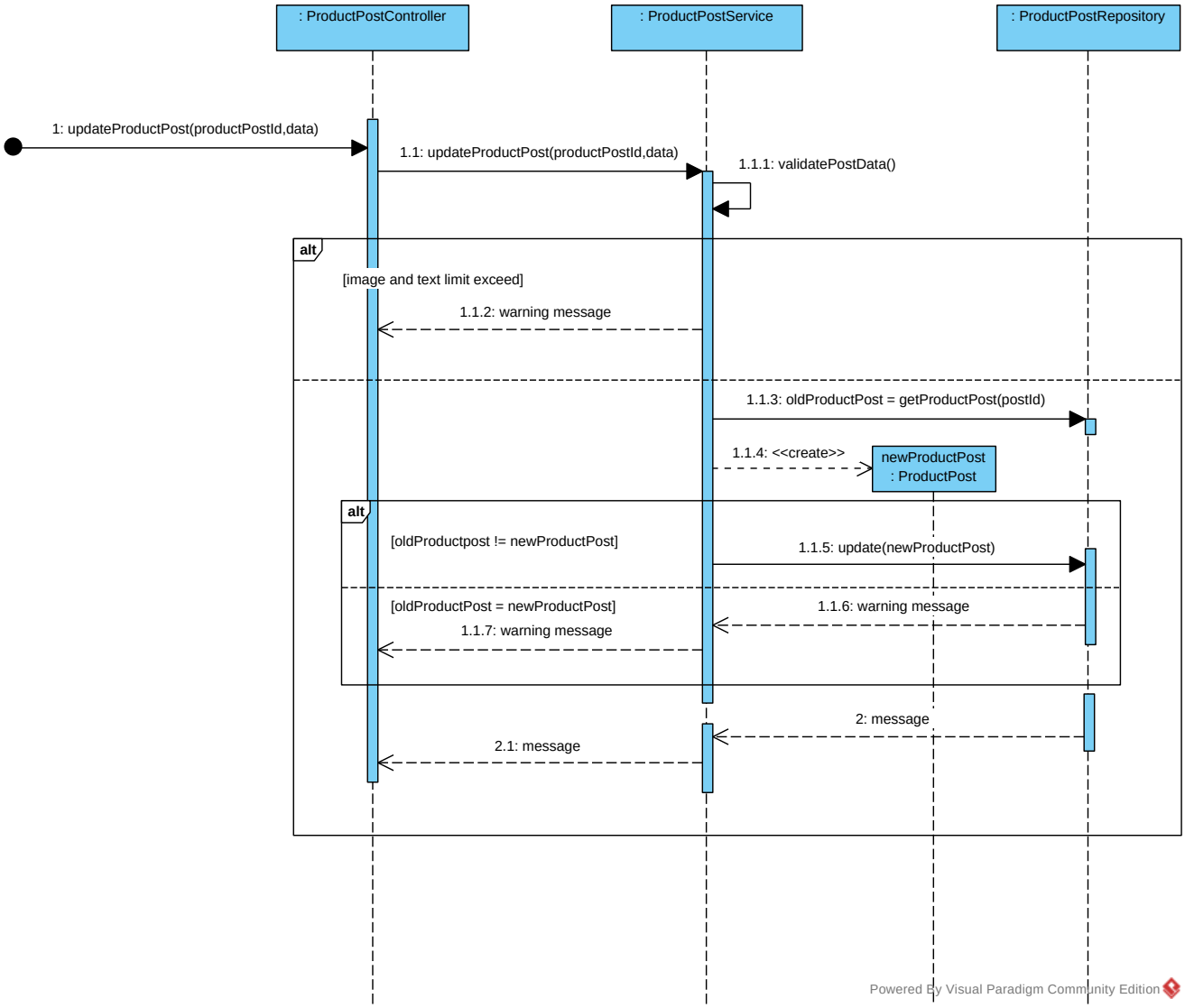
2.2.1 Operation Contract: banUser.

Use Case: Functional Requirement 8: Ban User.
Operation: banUser(userId)



2.2.2 Operation Contract: updateProductPost.

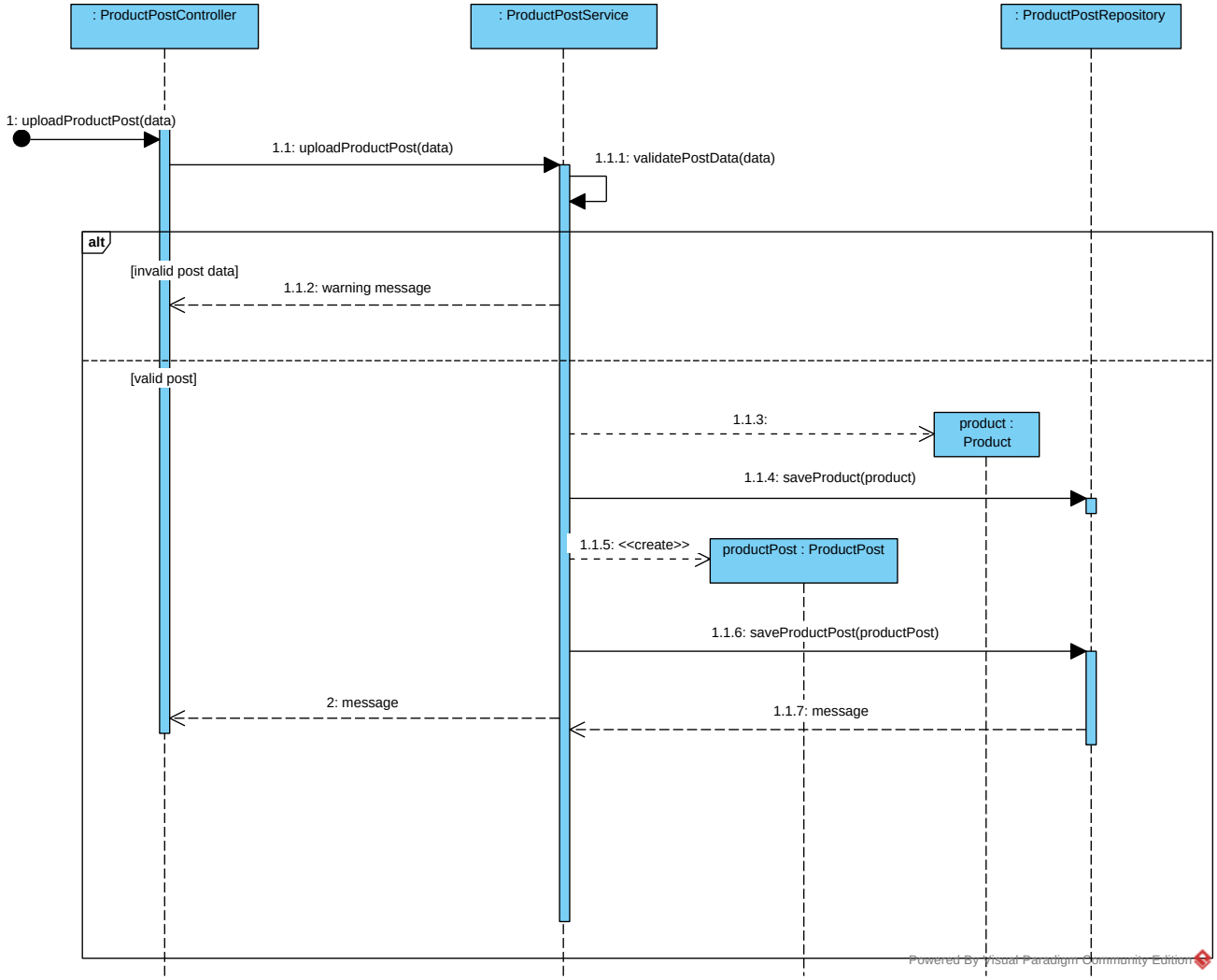
Use Case: Functional Requirement 9: Update ProductPost.
Operation: updateProductPost(productPostId, data)



2.3 Razwan Ahmed Tanvir

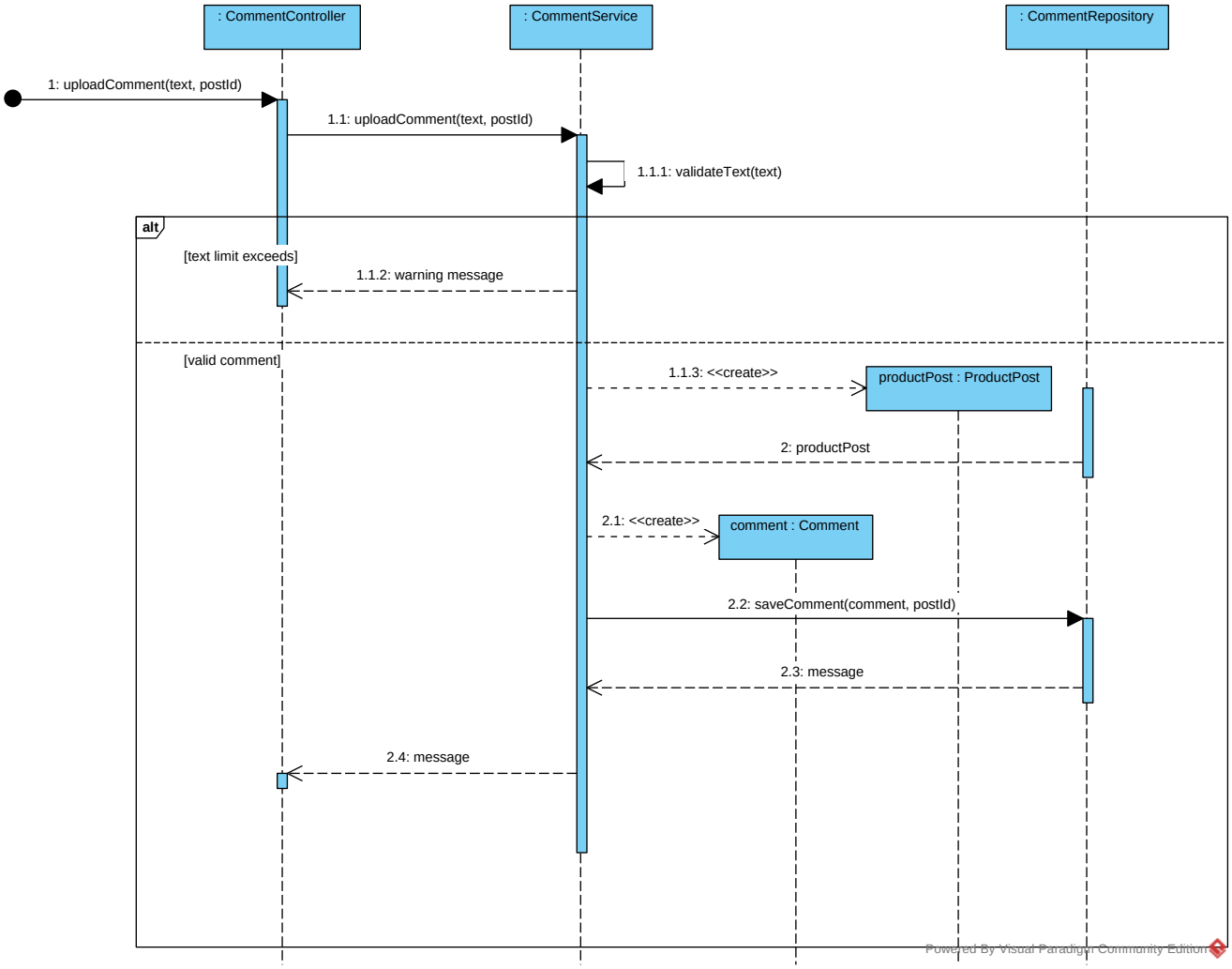
2.3.1 Operation Contract: uploadProductPost.

Use Case: Functional Requirement 4: Create ProductPost.
Operation: uploadProductPost(data)



2.3.2 Operation Contract: uploadComment.

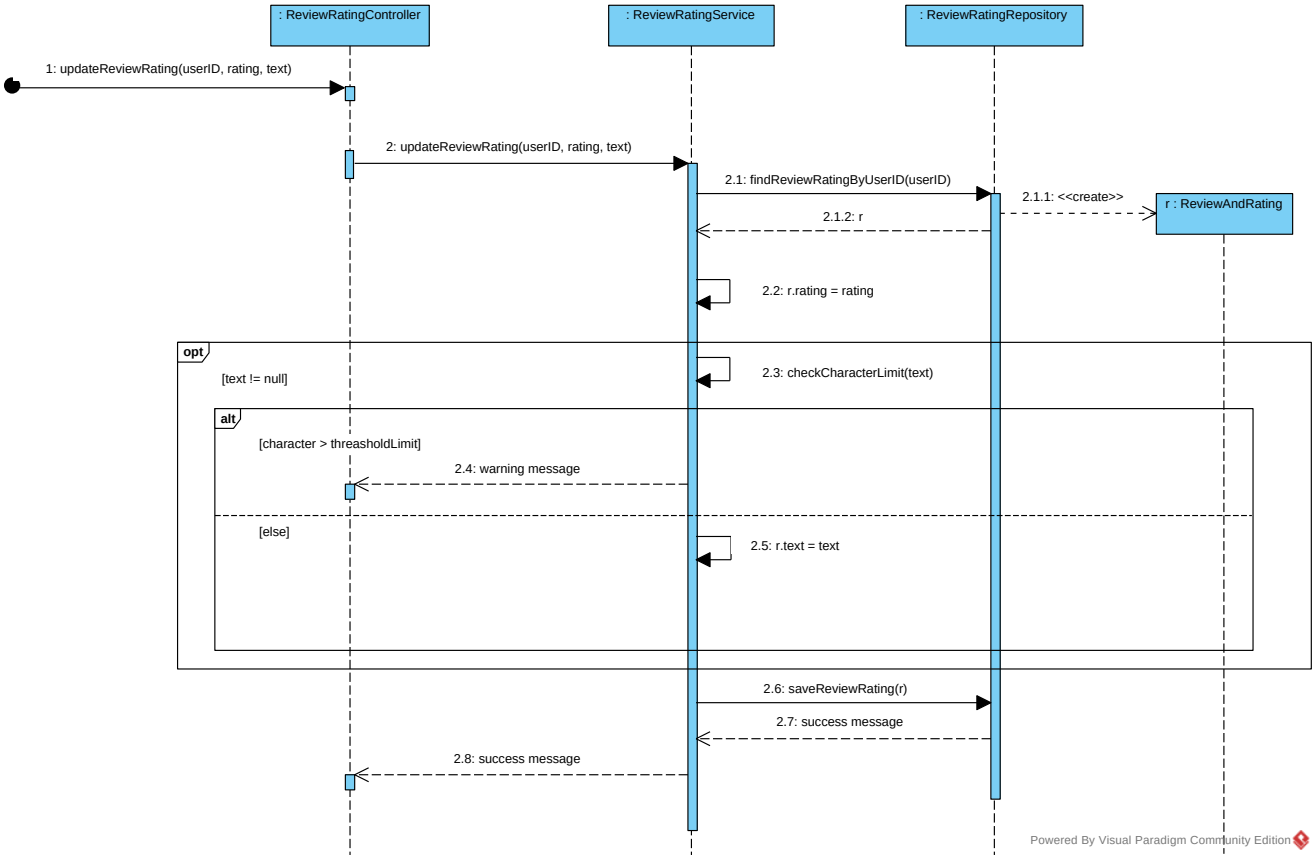
Use Case: Functional Requirement 5: Comment of ProductPost.
Operation: uploadComment(text, postId)



2.4 Swapnil Saha

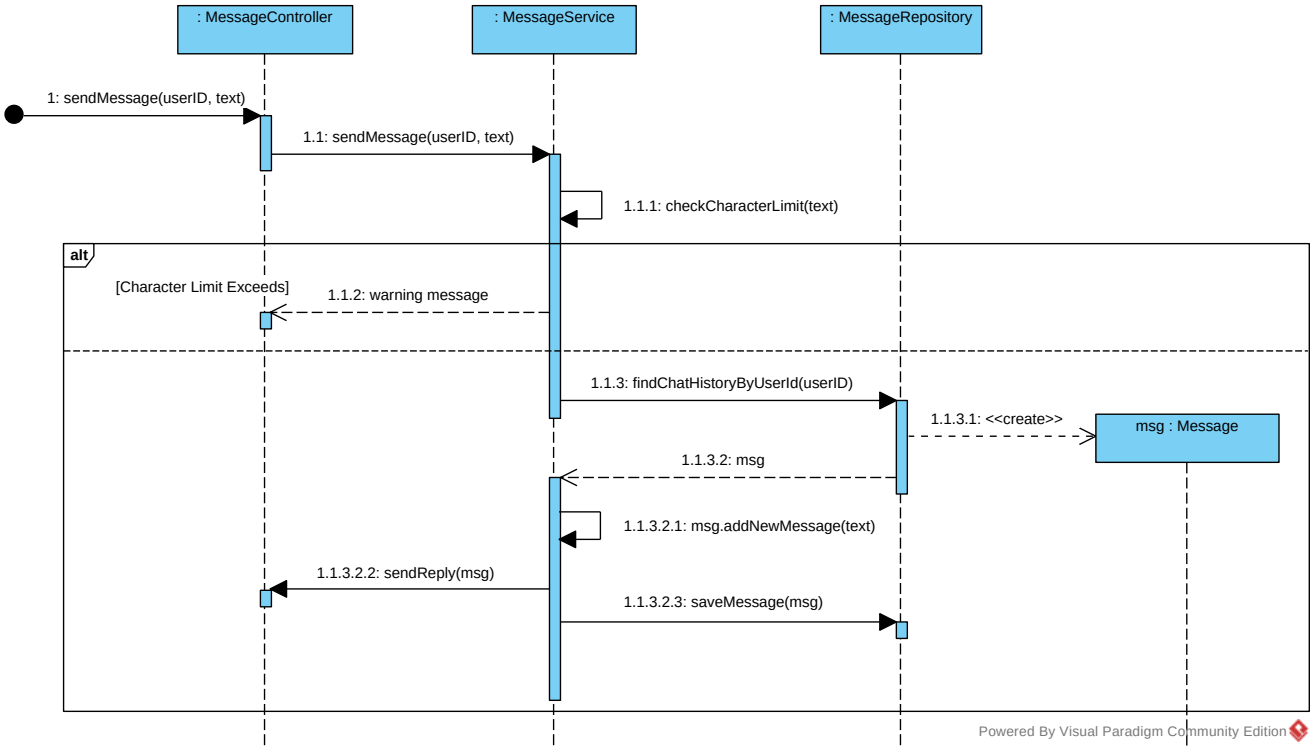
2.4.1 Operation Contract: updateReviewRating.

Use Case: Functional Requirement 13: Review And Rating.
Operation: updateReviewRating(userID, rating, text)



2.4.2 Operation Contract: sendMessage.

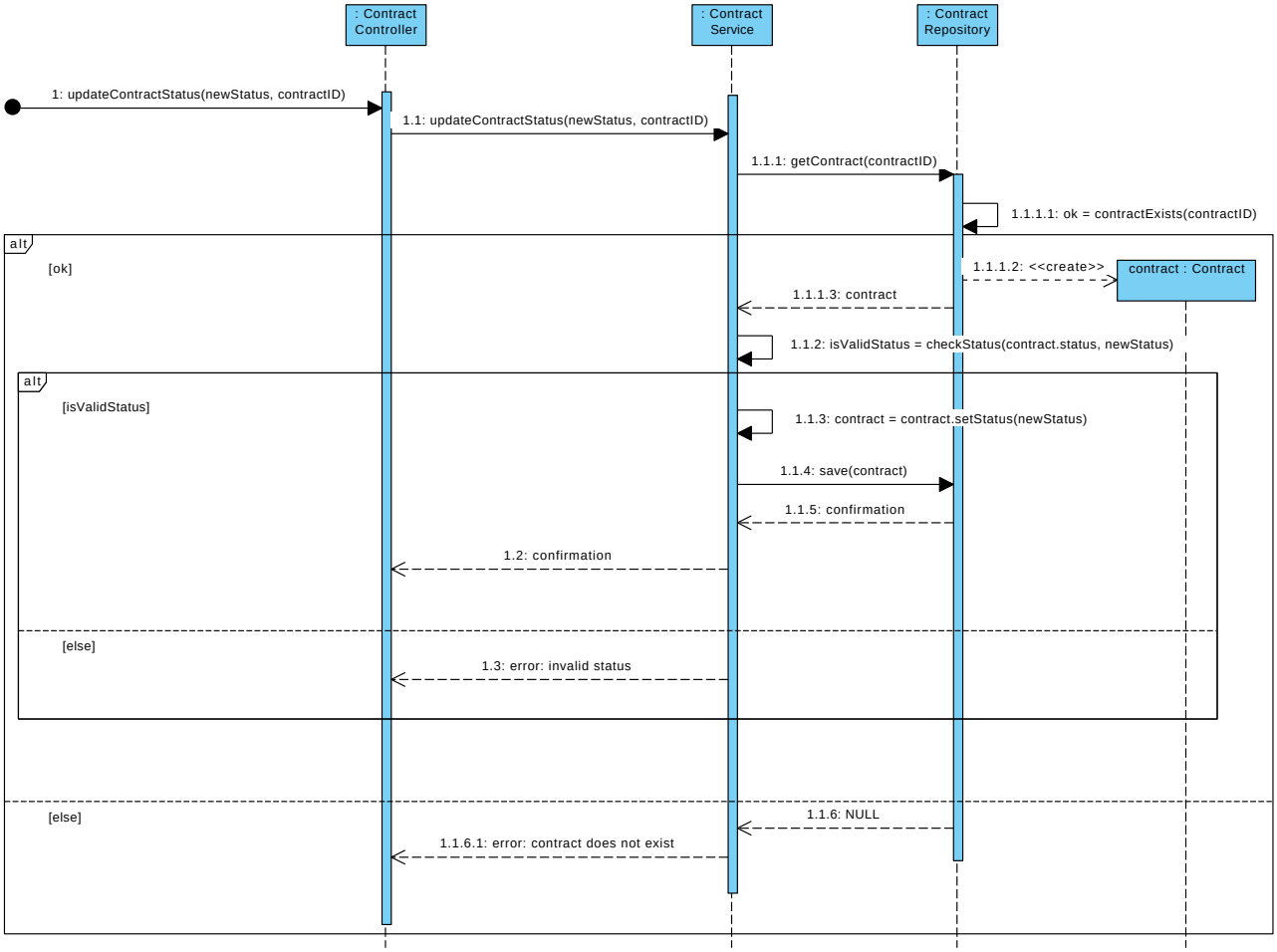
Use Case: Functional Requirement 14: Peer-to-peer Chat.
Operation: sendMessage(userID, text)



2.5 Tonni das Jui

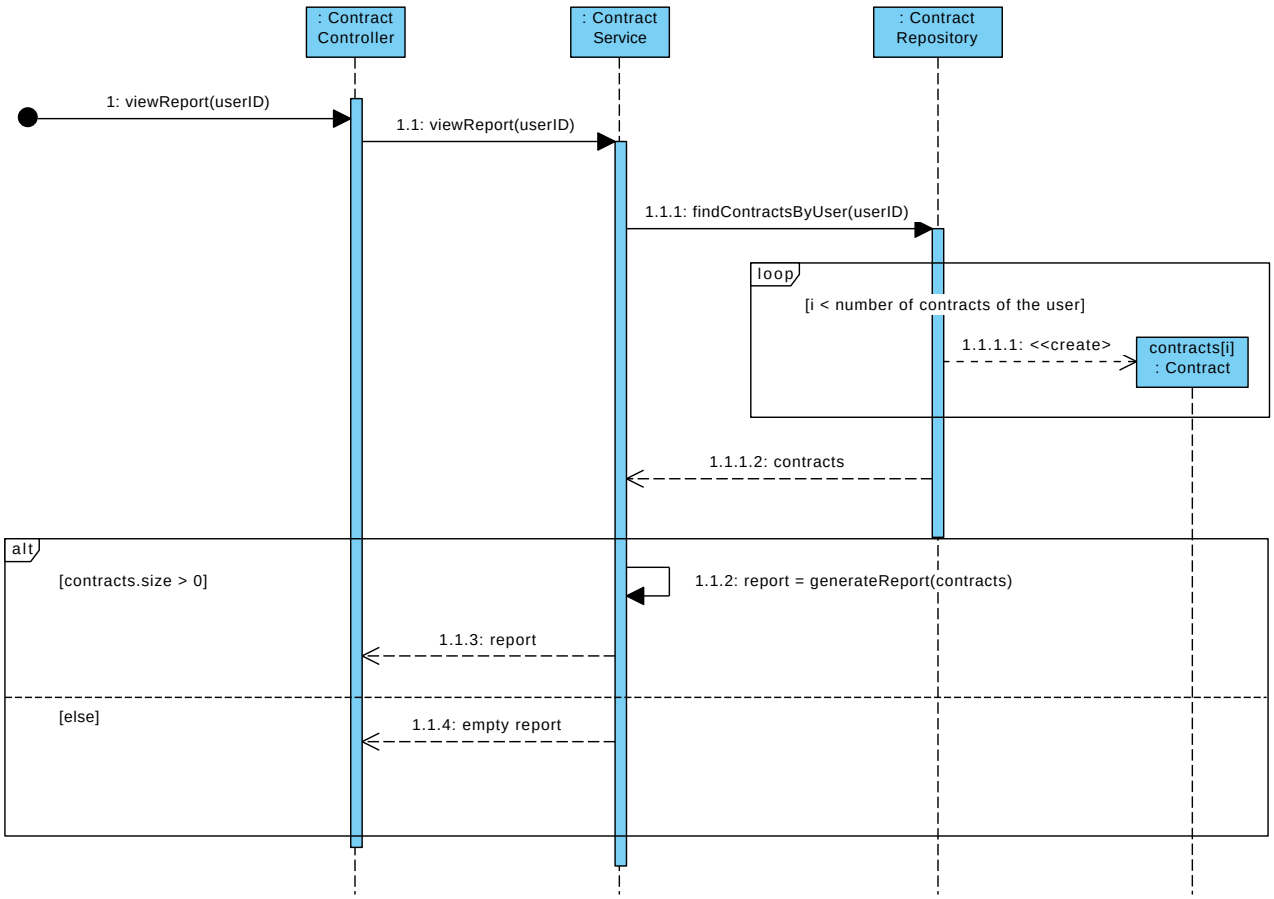
2.5.1 Operation Contract: updateContractStatus.

Use Case: Functional Requirement 1: Product Tracking.
Operation: updateContractStatus(newSatus, contractID)



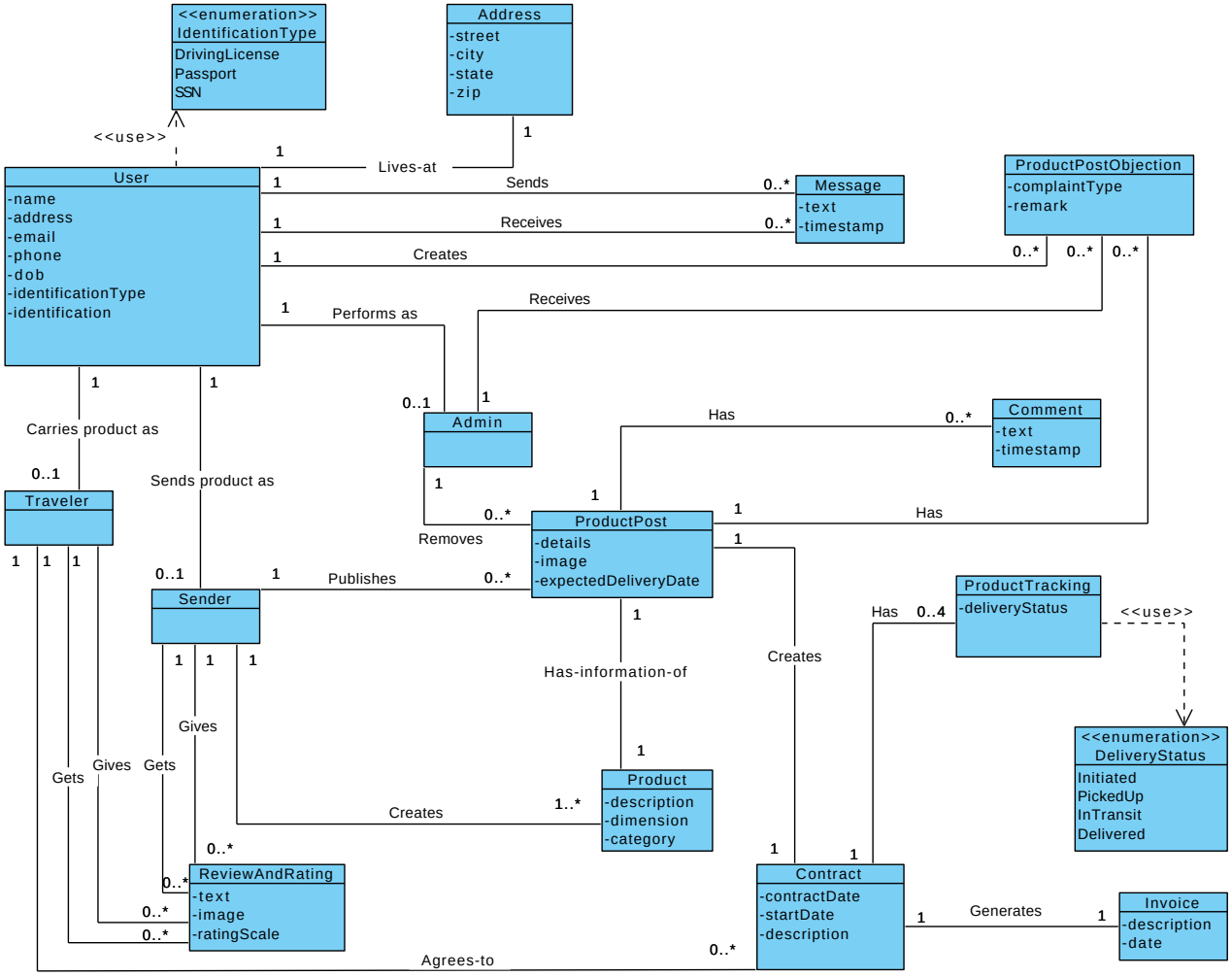
2.5.2 Operation Contract: viewReport.

Use Case: Functional Requirement 2: Contracts of User.
Operation: viewReport(userID)



3 Identification of GRASP patterns

3.1 Project Domain Model



3.2 GRASP pattern: Information Expert

1. *User* - To assign the responsibility of knowing how many productPosts a user has, how many contracts a user has, how many reviews and ratings the user has, we look for a class that has the information to determine these. *User* class knows all these along with all the *Sender*, *Traveler*, *Admin*, and *ReviewsAndRating* instances. *User* is an information expert for these responsibilities.

2. *User* - To know about the streets, states, cities, and zip codes of a user, it is necessary to know about all the *Address* instances. *User* is an information expert for this responsibility.

3. *ProductPost* - To know about the user's opinion about a *ProductPost*, it is necessary to know about all the *Comment* instances. *ProductPost* is an information expert for this responsibility.

4. *Contract* - *Contract* is the information expert on answering *Contract* description (which users are in this agreement, what is the product, what is the status of the delivery currently, etc.). It is necessary to know about the *Product* instance of a *ProductPost* and *User* instances and *ProductTracking* instances. A *Contract* instance contains these. A *Contract* is an information expert for this responsibility for knowing what is the product of this contract and who are the sender and traveler of this contract.

5. *ProductTracking* - To fulfill the responsibility of knowing the status of any product delivery, a *ProductTracking* needs to know *DeliveryStatus*. The *ProductTracking* is the information expert on answering its *DeliveryStatus*.

To fulfill the responsibility of knowing and answering the user's total number of contracts and their states, three responsibilities will be assigned to three design classes, *Contract*, *ProductPost*, *ProductTracking*. The fulfillment of this responsibility requires information that is spread across these classes of objects. So, they are the "partial experts" who will collaborate on the task.

3.3 GRASP pattern: Creator

1. *ProductPost* is a creator of *Product*.
   Whenever a sender creates a productPost, the system will create a *Product* instance by extracting the information from ProductPost. Therefore, *ProductPost* class is responsible for creating a *Product* instance, hence the creator.

2. *ProductPost* is a creator of *Contract*.
   Whenever a sender creates a productPost system will create an initial *Contract* instance for a particular productPost using the information from ProductPost. So, the *ProductPost* class is responsible for creating a contract.

3. *User* is a creator of *ProductPost* and *ProductPostObjection*.
   A user creates these class instances, hence *User* is a creator.

4. *Contract* is a creator of *Invoice*.
   A *Contract* instance contract generates an *Invoice* instance using the data from the class.

3.4 GRASP pattern: Low Coupling

1. Coupling exists between *ProductPost* and *Product*. For creating an instance of *ProductPost*, we will have attributes that refer to *Product* as productPost will contain the details about the dimension, description, and category of the product. *ProductPost* also will have attribute of *Contract*. For creating a *Comment* instance, a parameter with productPostID is required and an attribute referring to *Comment* instances is in *ProductPost*. This information is collaborative and simple to reuse, so, we are considering them low-coupling.

2. Low Coupling exists between *ProductPostObjection* and *ProductPost*. *ProductPostObjection* will have attribute of *ProductPost*.

3. Coupling exists between *User* and *Address*. There will be an attribute referring to multiple instances of *Address* as a user can have multiple addresses. This dependency is easier to follow and thus refers to low coupling according to our consideration.

4. Coupling exits between *Contract* and *ProductTracking*. *Contract* will keep information about some attributes from *ProductTracking* and *ProductPost* which is why low coupling exists between them.

3.5 GRASP pattern: High Cohesion

1. *ProductPost* has low cohesion as it has the responsibility to create both product and contract. However, our system suggests that a product and contract must be created when a user creates a ProductPost. Hence we can not remove the product and contract creation responsibility from ProductPost.

2. *User* has high cohesion because a user has moderate responsibility and it collaborates with other classes to fulfill a task. *User* class only collaborates and delegates.

3.6 GRASP pattern: Controller

Here are some of the controller classes we initially identified.

1. *ProductTrackingController*: *ProductTracking* controller handles updating the status of the product delivery. It routes requests to *Contract* entity and interacts with the view.
2. *ContractController*: If an *User* instance has an identificationType of "Admin", it can get any other registered user's contract details with productTracking status. So, *ContractController* controller handles this task. In addition, it handles updating *Contract* instance attributes.
3. *ProductPostController*: This controller routes *User* instances' request to create a *ProductPost* instance with small and specific operations. Furthermore, it handles the situation where any *User* instance requires searching for a specific *ProductPost* instance with possible filtering operations.
4. *ReportProductPostController*: It routes an *User* instances' request to store a complain against a *ProductPost* instance.
5. *ChatController*: This controller class handles peer-to-peer chat between two *User* instances.

3.7 GRASP pattern: Polymorphism and Dynamic binding

In our system model, we have a *User* class that will have a *getReport*() method. But the sub-classes *Admin*, *Traveler*, and *Sender* will implement it differently. For example, for *Sender*, it will give the created *ProductPost* related information. For *Traveler*, it will give the delivered *ProductPost* related information. If we have $User\ u = new\ Traveler()$ and call $u.getReport()$, we will use the *getReport*() implementation of the *Traveler* class. In this way, we can avoid writing conditional statements.

3.8 GRASP pattern: Pure Fabrication

For responsibility assignment, the Expert suggests that *User* instances are persistent and responsible for many user-related tasks. However, we will have to place a lot of work related to DB operations to the object *User* which is incohesive. *User* class is then coupled to the DB interface and it does not bring reusability. We can create a new class (e.g. SpringBoot Repository) responsible for DB persistence to ensure high cohesion and reusability. This way the responsibilities are handled by an artificial class (e.g. UserRepository) with a very specific set of related tasks (DB operations) to support reuse and high cohesion.

4 Object Constraint Language (OCL)

4.1 Three main OCLs for our system

4.1.1 ProductTracking constraint

Constraint: The deliveryStatus can be "Initiated", "Picked-up", "In-transit" and "Delivered". The deliveryStatus can only be updated before the contract expires. The deliveryStatus of ProductTracking can only be updated with a forwarding status.

For example, if the previous status is "In-Transit", the next status can only be "Delivered", but it cannot be "Picked-up" which was the previous status.

```
context ProductTracking::updateStatus(newStatus : String)

let allStatus := Sequence{Initiated, Picked-up, In-Transit, Delivered}
let i1 = allStatus.indexOf(newStatus)
let i2 = allStatus.indexOf(self.deliveryStatus@pre)
let updateDate = Date.now()

pre: allStatus.includes(newStatus) and i1 > i2
     and updateDate.isBefore(self.contract.endDate)
post: self.deliveryStatus = newStatus
```

4.1.2 ProductPost moderation constraint

Constraint: Only the admin can moderate a ProductPost if the number of complaints is greater than the threshold. If the moderator accepts the complaints, the ProdcutPost should be on hold.

```
context ProductPost::moderate(moderatedBy : User, accepted : Boolean)

pre: moderatedBy.userType = #ADMIN
     and self.complaints -> size() > threshold
post: accepted = True implies self.tag = OnHold
```

4.1.3 ReviewAndRating constraint

Constraint: Only users associated with a contract can review each other. Review can be submitted only after the contract end date. A user can not review himself. After submitting the review, rating of the rated user should be updated.

```
context ReviewAndRating::submitReview()

let currentRating = self.ratedUser.rating@pre
let currentReviewCount = self.ratedUser.reviewCount@pre

pre: self.rating >= 1 and self.rating <= 5
     and self.contract.endDate.isBefore(self.date)
     and self.ratedUser != self.submittedByUser
     and (self.submittedByUser = self.contract.sender
        or self.submittedByUser = self.contract.traveller)
     and (self.ratedUser = self.contract.sender
        or self.ratedUser = self.contract.traveller)
post: self.ratedUser.rating = (currentRating * currentReviewCount
        + self.rating) / (currentReviewCount + 1)
     and self.ratedUser.reviewCount = currentReviewCount + 1
```

4.2 Other OCLs for our system

We have also figured out some initial constraints for our system and we have listed them below for later implementation.

4.2.1 Admin constraint

Constraint: There must be at least one administrator.

```
context User

inv: User.allInstances() -> exists ( u | u.userType = #ADMIN)
```

4.2.2 ProductPostObjection constraint

Constraint: User can attempt a ProductPostObjection only once. Even after the Admin has resolved the complaint and decided to keep the ProductPost, the same user cannot object against the Poroduct-Post again.

```
context User::createProductPostObjection(pp : ProductPost)

pre: self.objectedProductPost() -> excludes(pp)
```

4.2.3 Contract constraint

Constraint: Contract start date has to be at least one day before the contract end date.

```
context Contract

inv: self.startDate.isBefore(self.endDate)
```

4.2.4 UserType constraint

Constraint: All Users must have the user type either ADMIN or TRAVELER or SENDER.

```
context User

inv: User.allInstances() -> forAll (p | p.userType = #ADMIN
                                        or p.userType = #TRAVELER
                                        or p.userType = #SENDER)
```

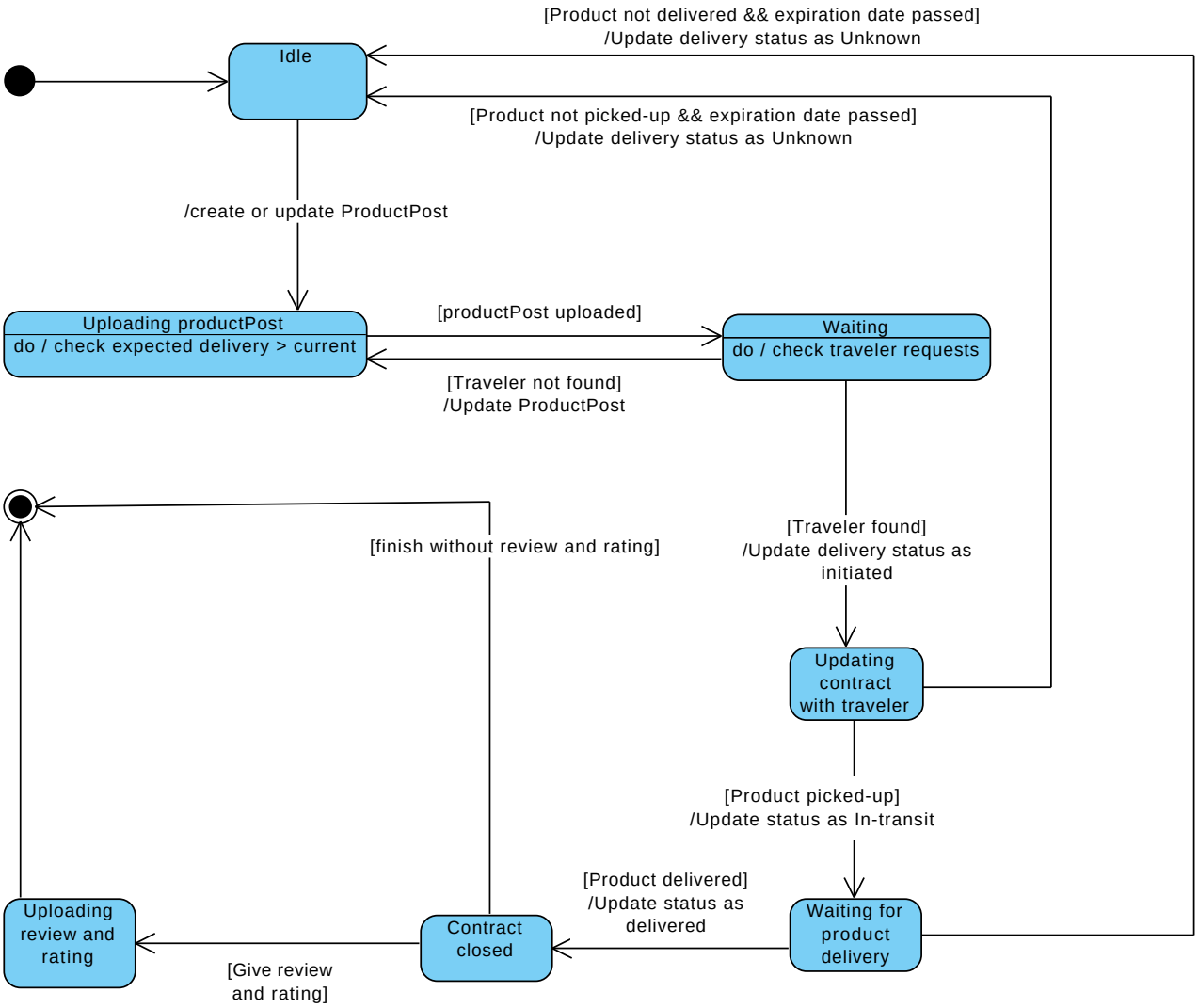
4.2.5 PromoteUser constraint

Constraint: Only Administrator can promote a user to an administrator if the target user is not already an administrator.

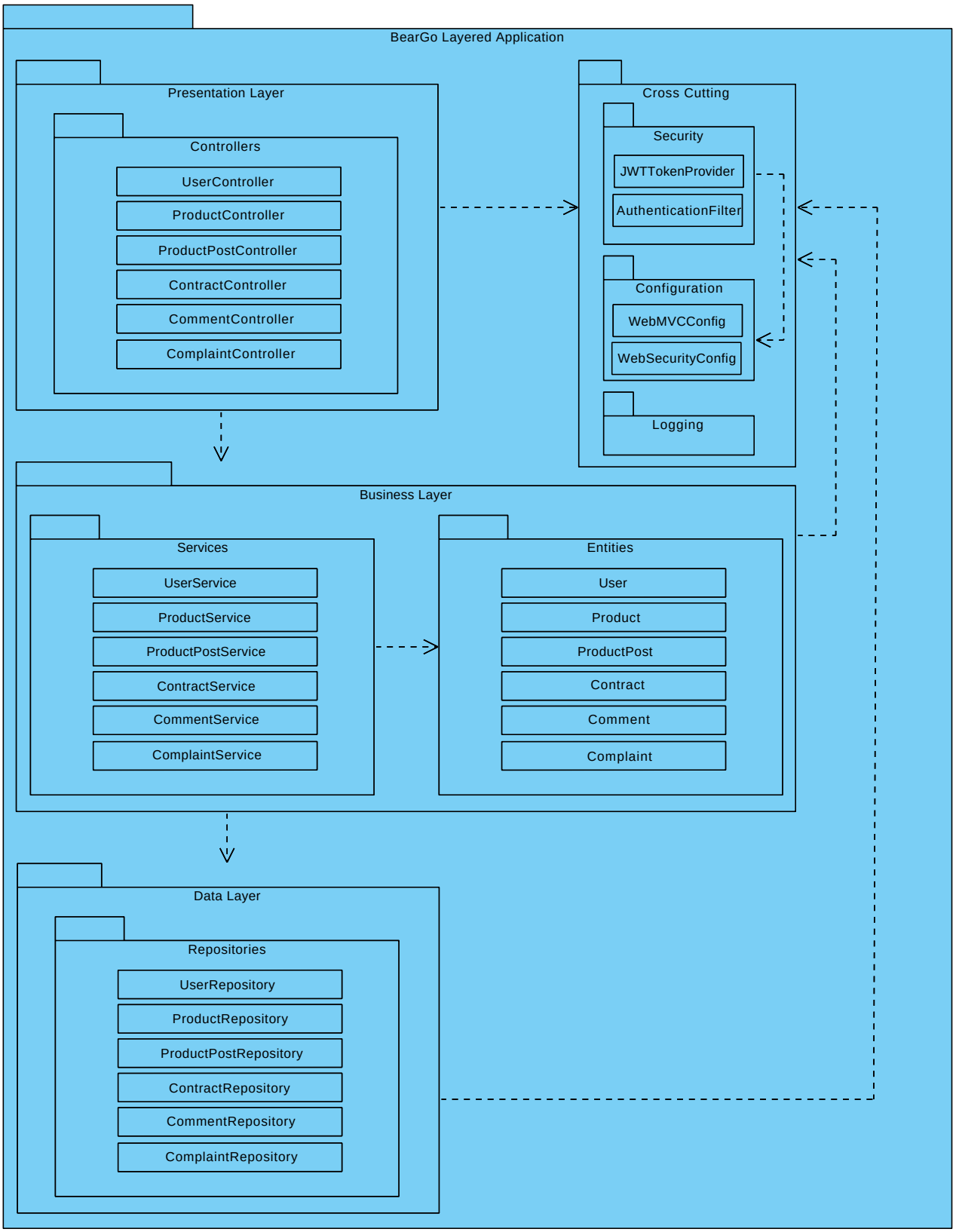
```
context User::promoteUser(t : User)

pre: self.userType() = #ADMIN and t.userType != #ADMIN
```

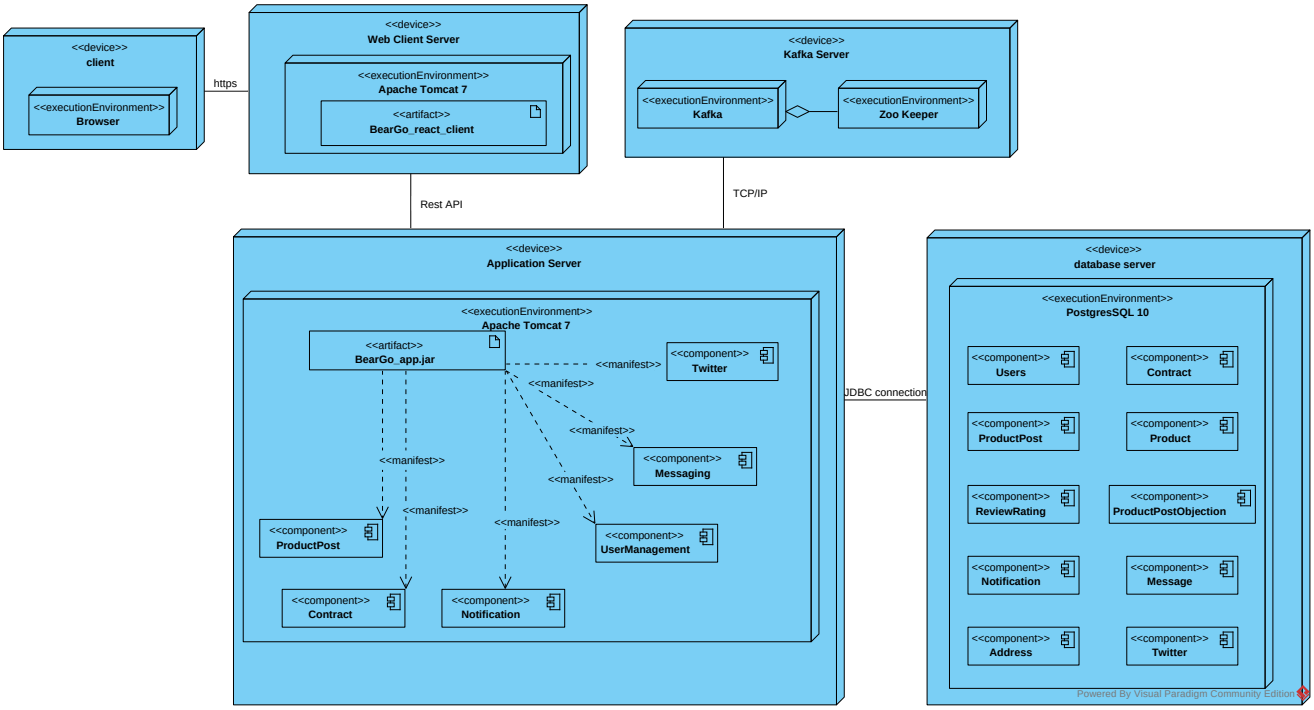
5 State Machine Diagram



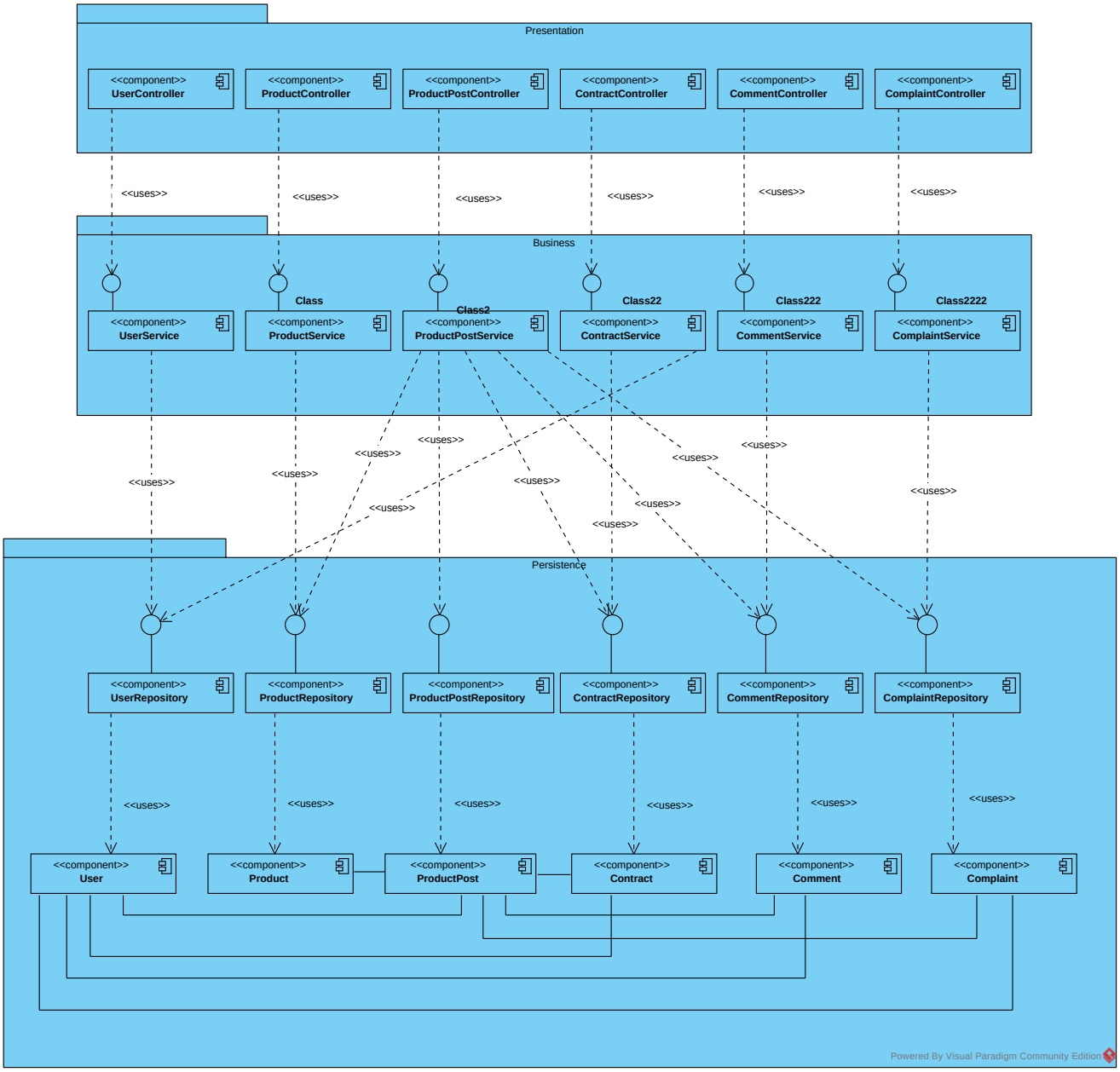
6 Package Diagram



7 Deployment Diagram

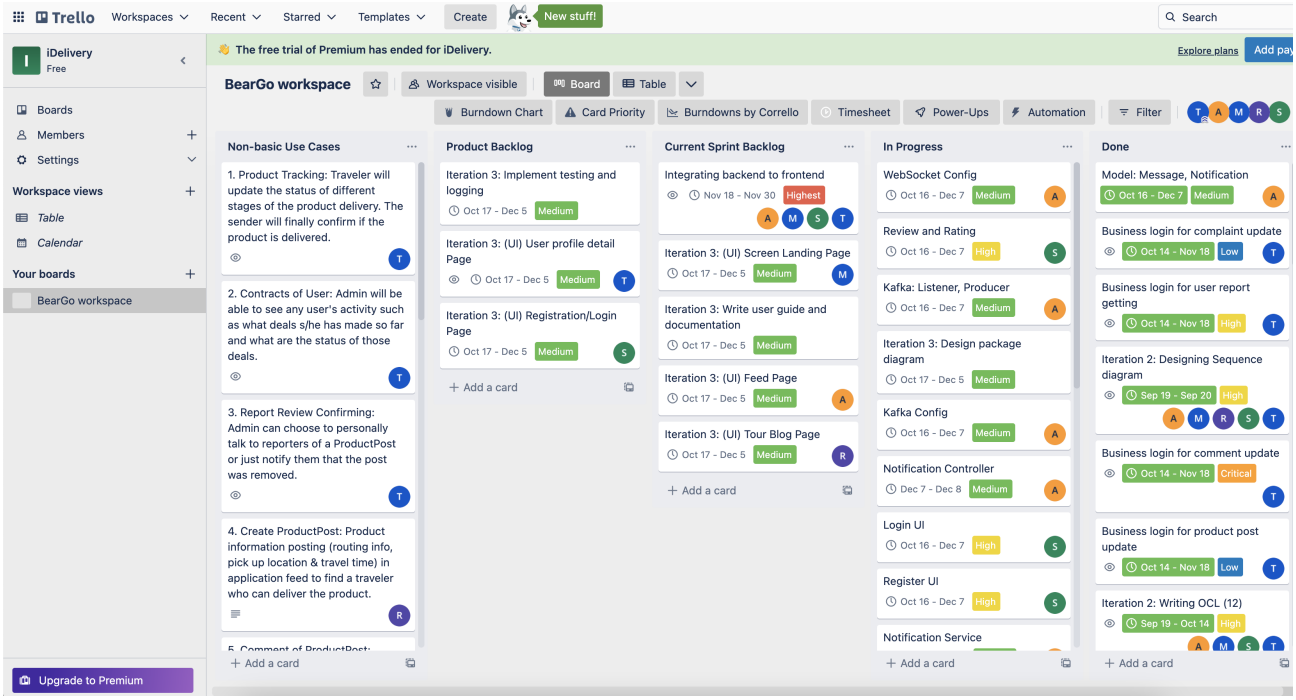


8 Component Diagram



9 Trello Updates

9.1 Trello Overview Overall



9.2 Trello Burndown Chart

9.3 Individual Trello Task View

9.3.1 AGM Islam

Task	Hours
Daily Scrum	11.25
Iteration 1: Create GitHub repository (1)	2
Iteration 1: Data Model(3)	1.67
Iteration 1: Deciding frameworks (backend and frontend) and create a skeleton project (6)	2
Iteration 2: Demo with CRUD REST API (12)	6.5
Iteration 2: Design component diagram (10)	9
Iteration 2: Design deployment diagram (6)	3
Iteration 2: Designing Sequence diagram	7.5
Iteration 2: Identify GRASP patterns (8)	5
Iteration 2: State Diagram (11)	4
Iteration 2: Writing OCL (12)	4
Kafka Config	2
Kafka: Listener, Producer	4
Message Controller	3.5
Message Service	4.5
Model: Message, Notification	2.5
Notification Controller	4
Notification Service	4
Security Config	5
Sprint Retrospective	10
Sprint Review	4.5
WebSocket Config	4
Grand Total	**103.92**

9.3.2 Maisha Binte Rashid

Task	Hours
API connection created from frontend to backend	3
API for Ban user created	1.75
Business logic for search post modified	2.25
Business logic modified for ban user	2.5
Create API for Productpost update	2.5
Create API for Search post	2.5
Create productpost design modified	2.5
Create Productpost page added	2.25
Daily Scrum	10
Frontend home page added	4.5
Homepage design modified	4.5
Iteration 2: Demo with CRUD REST API (12)	6.42
Iteration 2: Design component diagram (10)	10
Iteration 2: Design deployment diagram (6)	4
Iteration 2: Designing Sequence diagram	3
Iteration 2: Identify GRASP patterns (8)	5
Iteration 2: State Diagram (11)	5.25
Iteration 2: Writing OCL (12)	4.25
Login page added	0.5
Login page design modified	1
Notification bar design modified	2.5
Notification design added	3
Register page added	2
Register page modified	2.5
Sprint Retrospective	8
Sprint Review	4
Grand Total	**99.67**

9.3.3 Swapnil Saha

Task	Hours
BlogPost Creation	9.25
Daily Scrum	11.25
Iteration 1: Create GitHub repository (1)	2
Iteration 1: Data Model(3)	1.55
Iteration 1: Deciding frameworks (backend and frontend) and create a skeleton project (6)	1.83
Iteration 2: Demo with CRUD REST API (12)	11
Iteration 2: Design component diagram (10)	7.27
Iteration 2: Design deployment diagram (6)	3.5
Iteration 2: Designing Sequence diagram	5.75
Iteration 2: Identify GRASP patterns (8)	3.75
Iteration 2: State Diagram (11)	3.5
Iteration 2: Writing OCL (12)	3.5
Login UI	5.75
Register UI	6.75
Review and Rating	6.75
Sprint Retrospective	10
Sprint Review	5
Grand Total	**98.4**

9.3.4 Tonni Das Jui

Task	Hours
Authentication system	4.2
Business login for comment update	2.1
Business login for complaint update	3.9
Business login for product post update	2
Business login for user report getting	4
CRUD for complaint creation	4
CRUD for contract	2.3
CRUD for create comment	2.8
CRUD for create product post	3.7
CRUD for finding traveler	2.5
CRUD for promoting user	3.7
CRUD for updating complaint	2.1
CRUD for updating user profile	2.1
Daily Scrum	10
Iteration 2: Demo with CRUD REST API (12)	5
Iteration 2: Design component diagram (10)	9.4
Iteration 2: Design deployment diagram (6)	3.8
Iteration 2: Designing Sequence diagram	3.2
Iteration 2: Identify GRASP patterns (8)	5.3
Iteration 2: State Diagram (11)	4.1
Iteration 2: Writing OCL (12)	4
JWT tokenizer	3.8
Sprint Retrospective	8
Sprint Review	4
Grand Total	**100**

10 System Development Report

10.1 Project Backend

10.1.1 Repository Link

Our project is available https://github.com/tonnidas/BearGo

10.1.2 Backend demo video

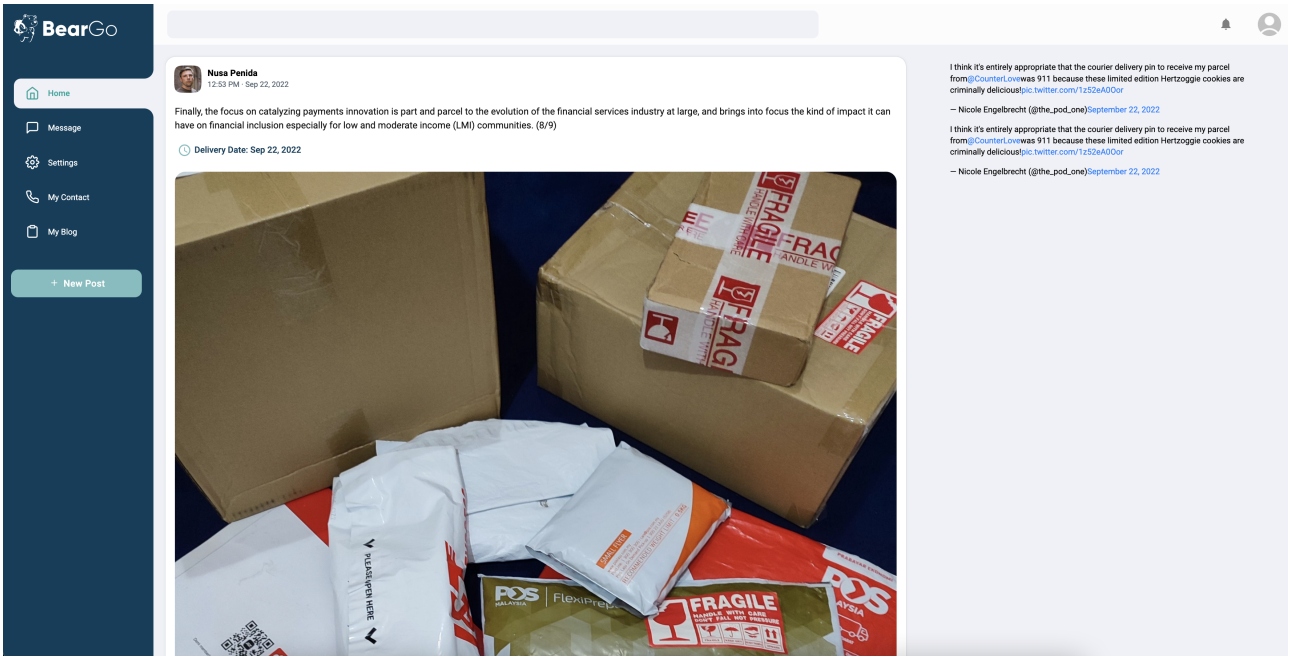
Our project backend demo video is available https://tinyurl.com/yv6tkbdm

10.2 Project Frontend

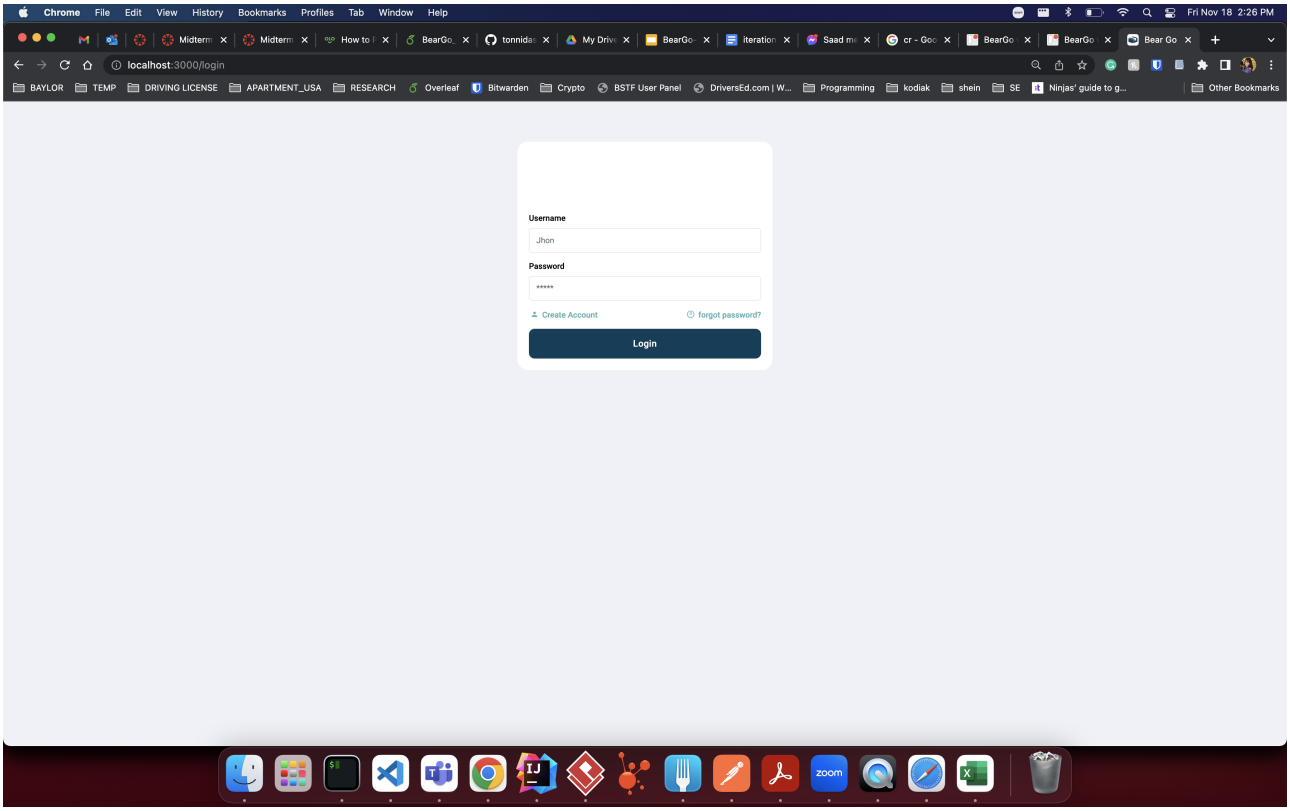
10.2.1 Frontend demo video

Our project frontend demo video is available https://tinyurl.com/25w8xmp9

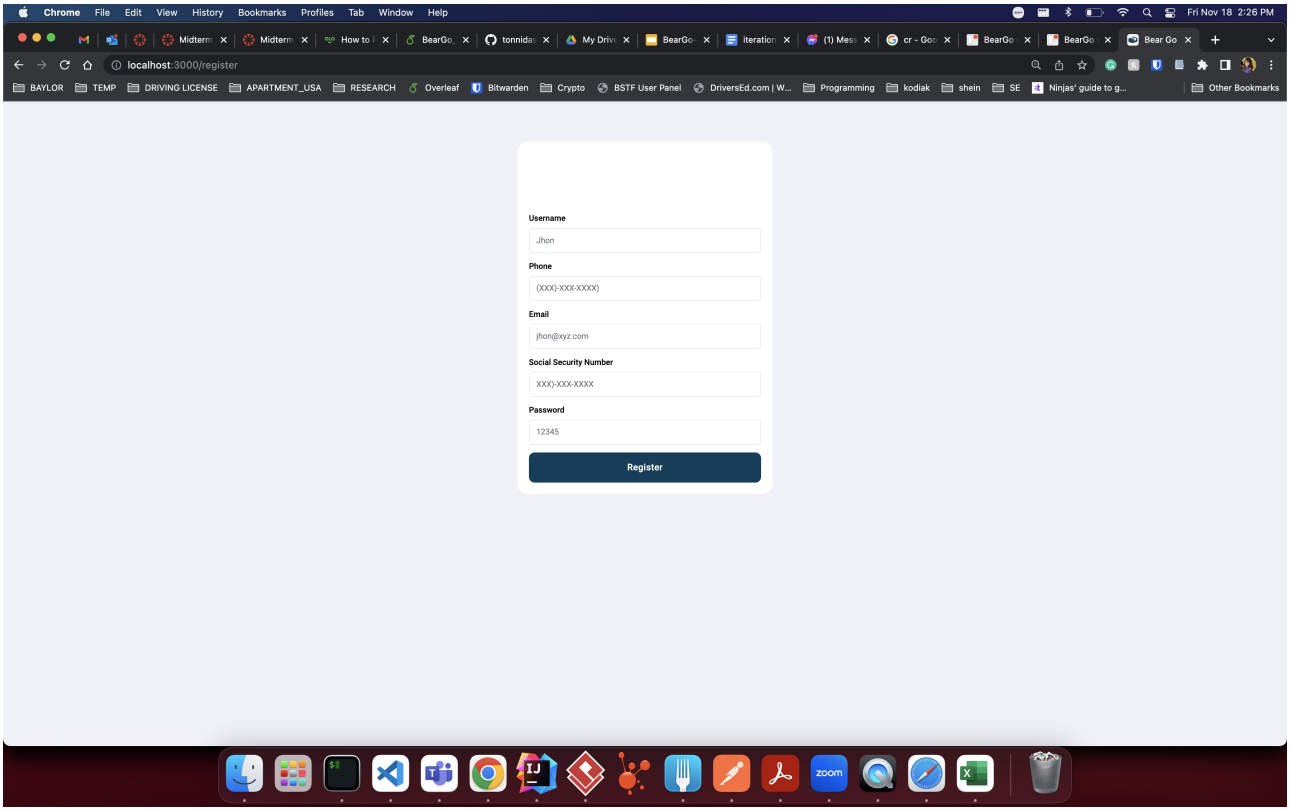
10.2.2 Frontend: Homepage View



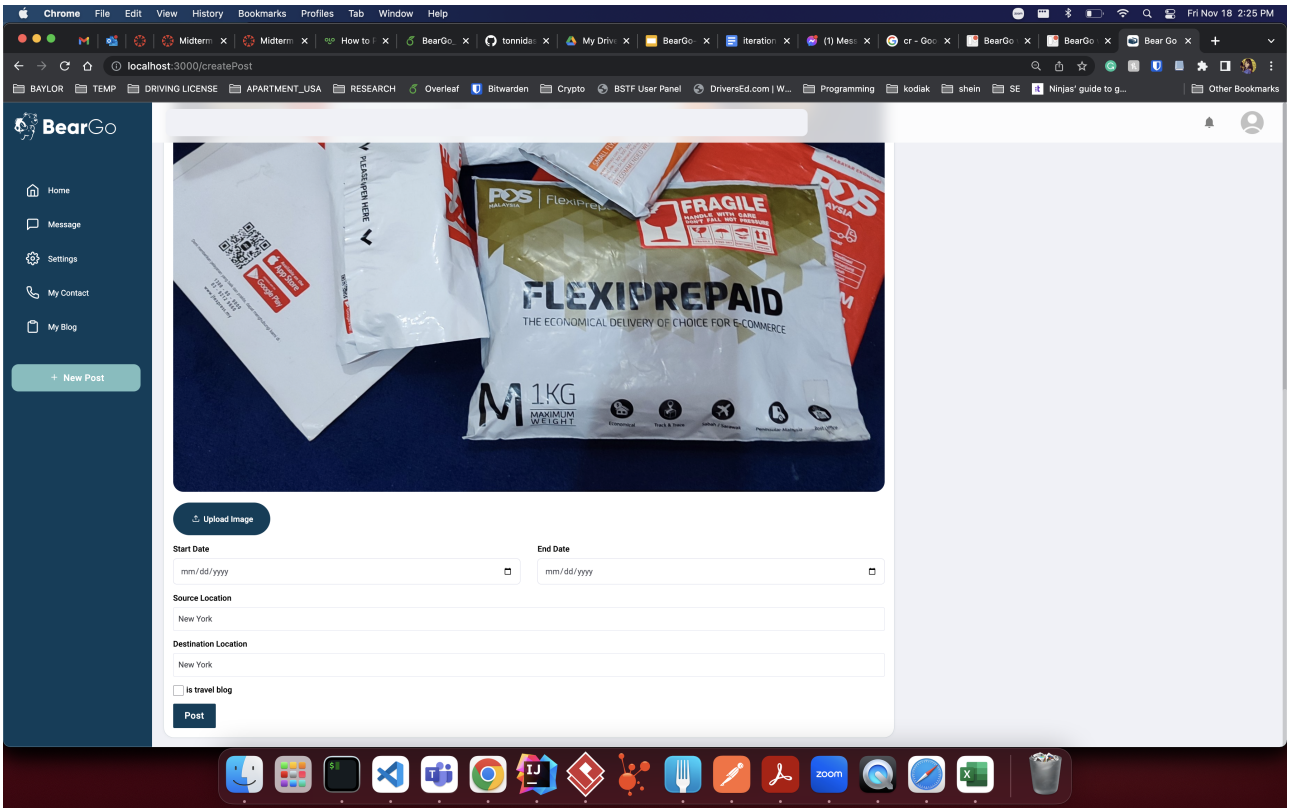
10.2.3 Frontend: Log In View



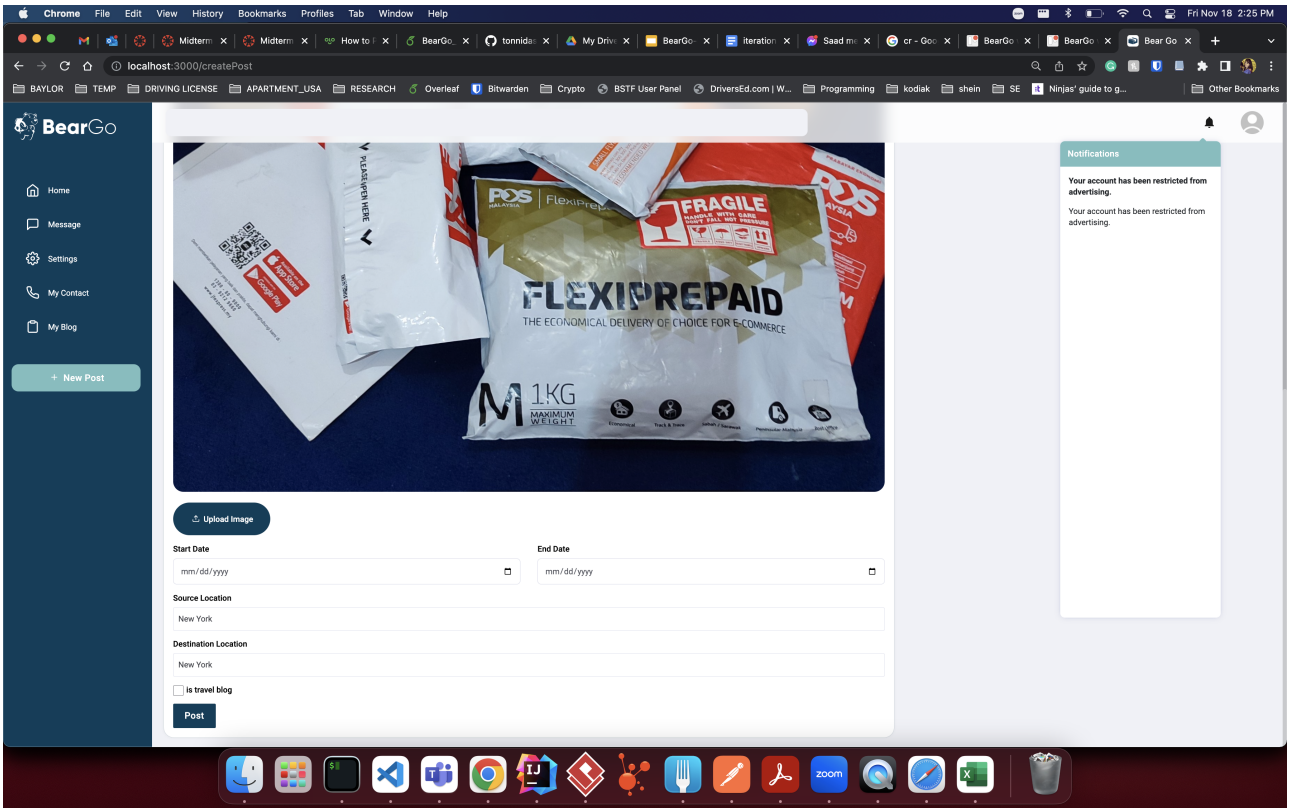
10.2.4 Frontend: Registration View



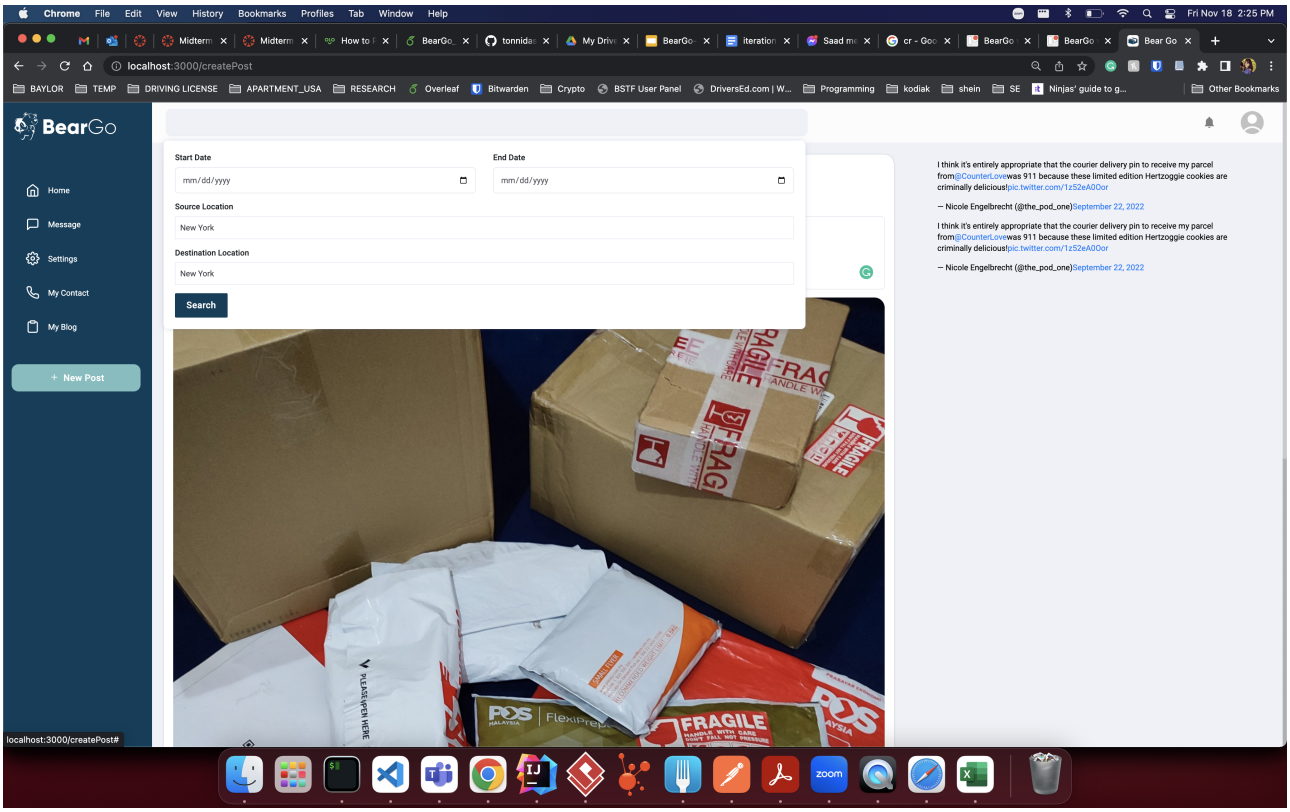
10.2.5 Frontend: Product Post Creation View



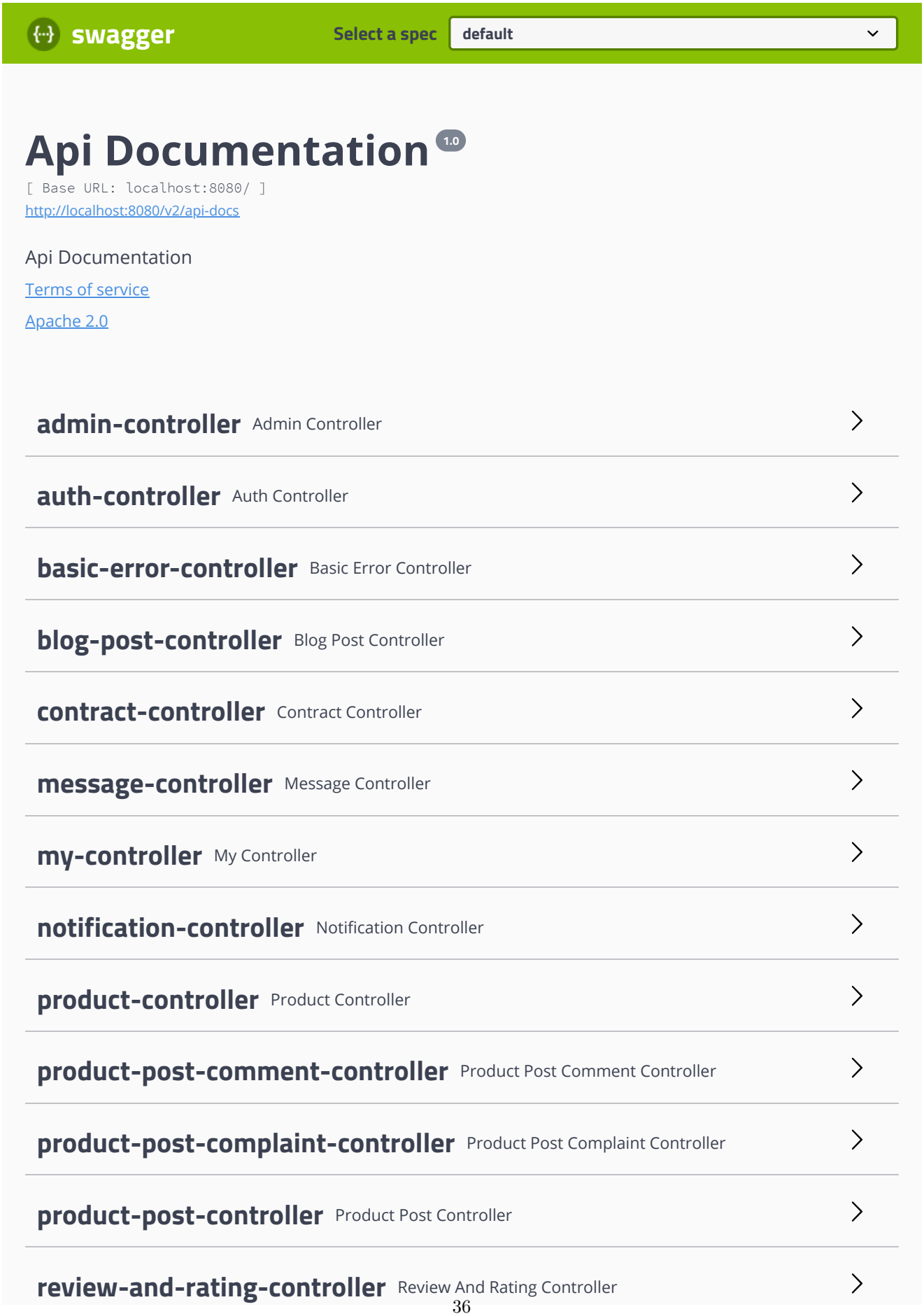
10.2.6 Frontend: Notification View



10.2.7 Frontend: Search View



10.3 Project Swagger API



user-controller User Controller

Models

- Address
- BlogPost
- Contract
- GrantedAuthority
- Message
- ModelAndView
- Notification
- Product
- ProductPost
- ProductPostComment
- ProductPostComplaint

ReviewAndRating ›

User ›

UserContracts ›

UserDetails ›

View ›