Iteration 3

BearGo

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.

2 Summary of proposed Functional Requirements

We initially planned to implement these main features in our project. We have separated our tasks to each group member according to these features and implemented them in diagrams accordingly.

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 is the status of those deals.
3. Report Review: Admin can review and choose to chat with reporters of a ProductPost or notify them that the post was removed.
4. Create ProductPost: Product information posting (routing info, pick-up location & travel time) in the 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.

3 Implemented features in brief

We are listing the features that we have implemented for admin and user panel and added them to give in-anutshell view of features.

3.1 User Panel

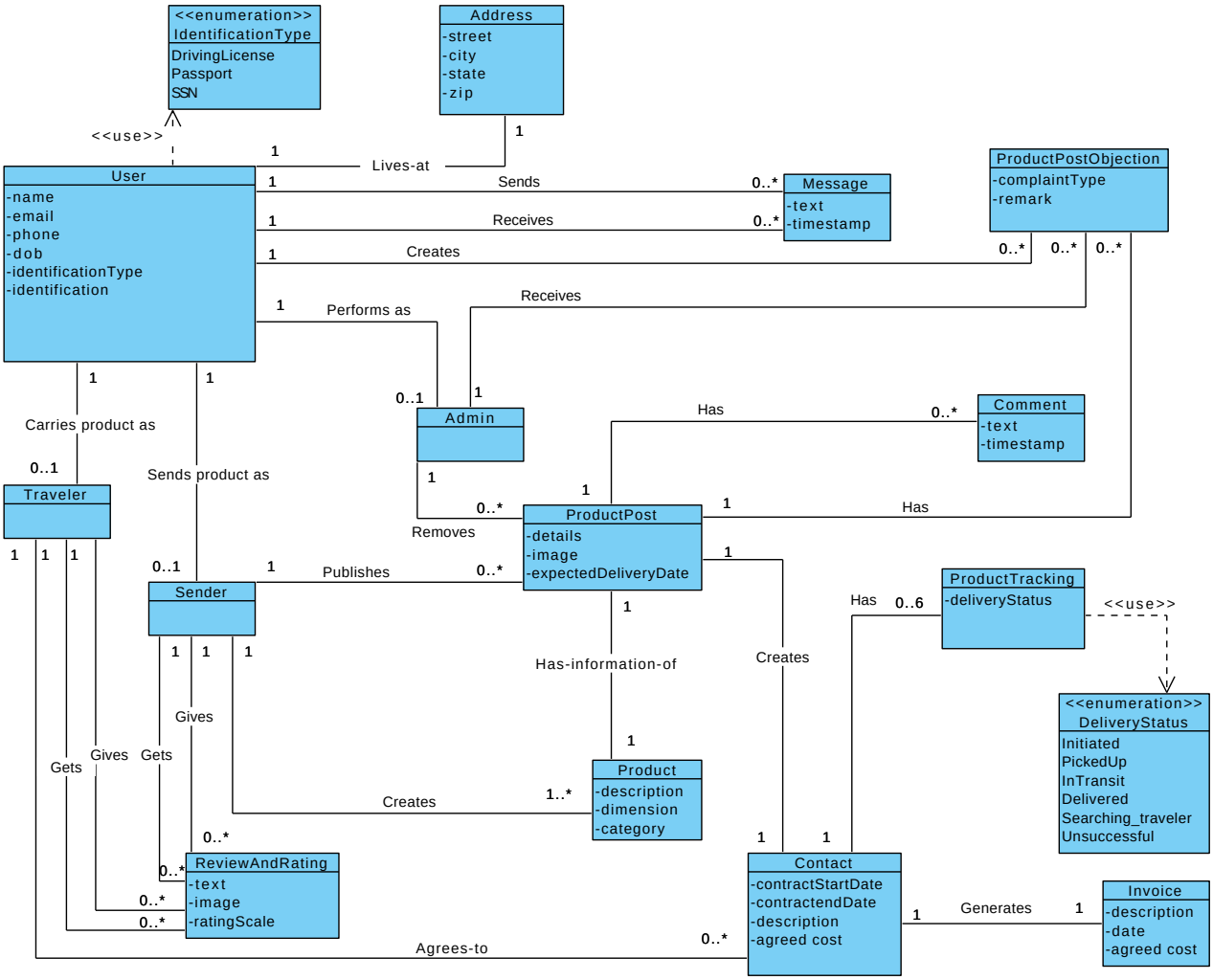
1. User registration and login.
2. Email verification during registration.
3. Reset forgotten password.
4. Updating profile information, profile picture.
5. Deactivating your account.
6. Create ProductPost with necessary info.
7. Modify productPost.
8. View all of user contracts.
9. Update product tracking status.
10. show interest to be traveler.
11. Pick a traveler from list of interested user.
12. Write comment.
13. Complain against a ProductPost.
14. Complain against a user.
15. View notifications.
16. Rate and review a user.
17. Messege a user.
18. Create BlogPost.
19. Search ProductPost using some filters.
20. Search a user.

3.2 Admin Panel

1. Promote a user to be admin.
2. View who are the other admins.
3. View a user or product post complain.
4. resolve a user or product post complain by banning product post or user.
5. sending notification of resolving result (ignoring/blocking) to all complainees.

4 Analysis and design

5 Class Diagram



5.1 Identification of GRASP patterns

5.1.1 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.

5.1.2 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.

5.1.3 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.

5.1.4 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.

5.1.5 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.

5.1.6 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.

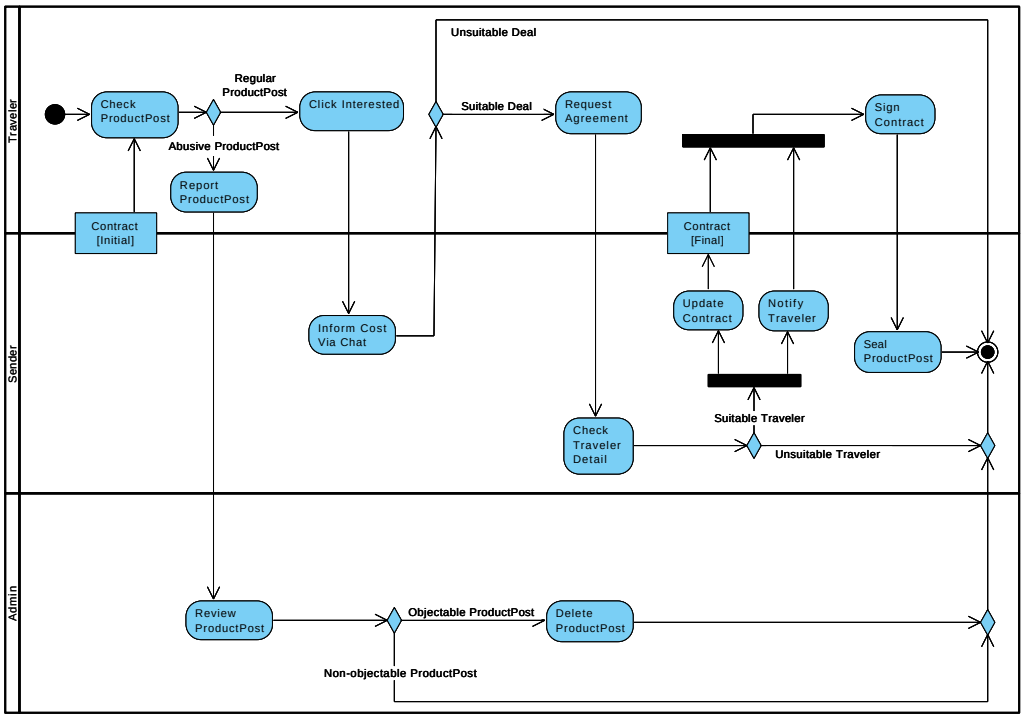
5.1.7 GRASP pattern: Pure Fabrication

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.

5.2 Activity Diagram

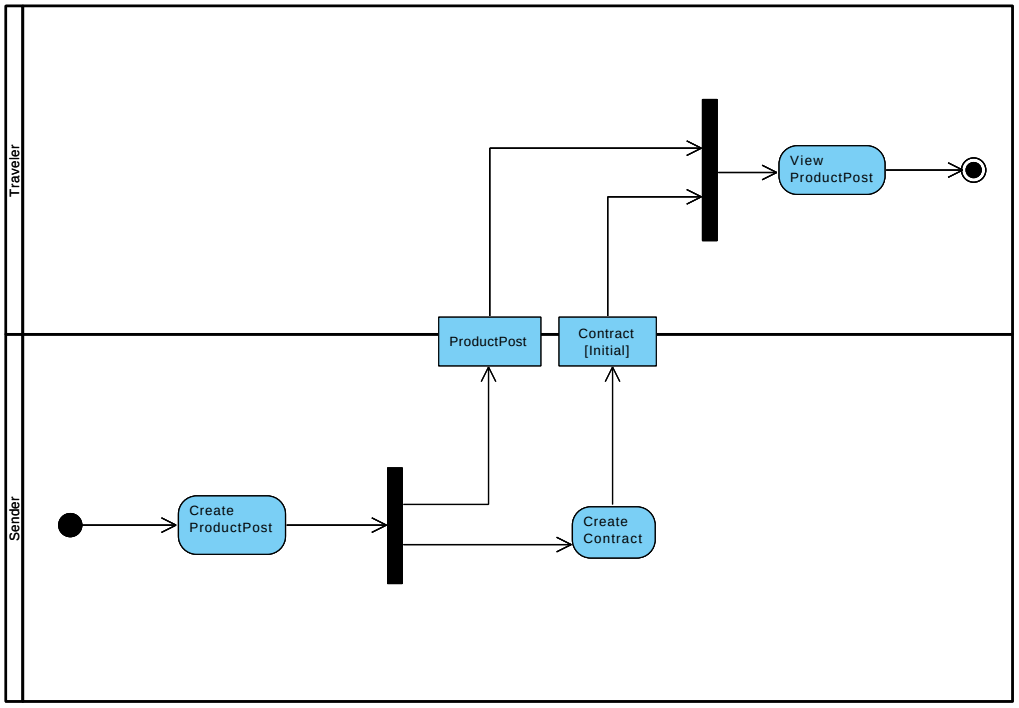
5.2.1 Activity diagram: Contract Creation

The activity diagram starts from creating ProductPost and contract. Multiple traveler may click "Interested" button and the sender communicates offering a deal via chat and seeks agreement. Traveler agrees if the deal is suitable and sender updates the contract status as "Initiated" so that this post is not showed to the feed anymore. However, a traveler can also complain if any content of the product post is inappropriate. An admin reviews the post to see if he will block the post or ignore the complain.



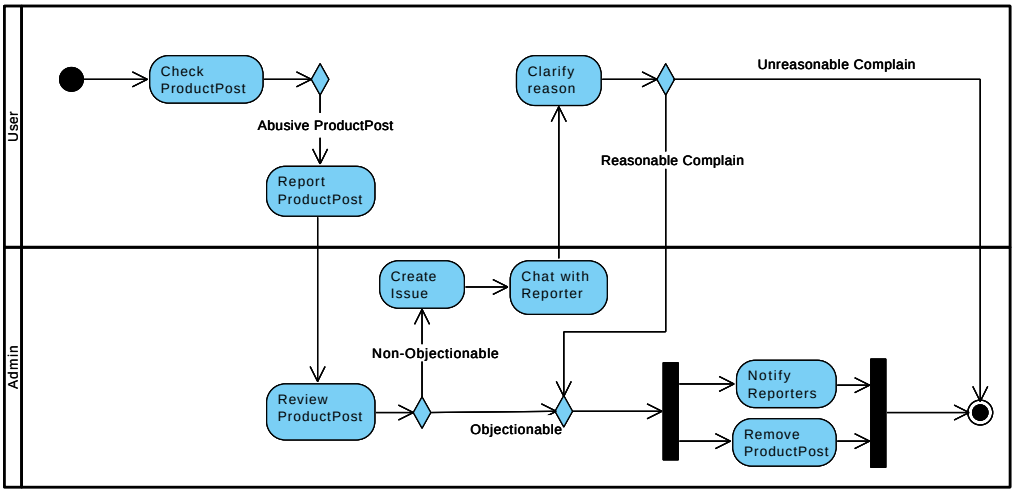
5.2.2 Activity diagram: Publishing ProductPost

The activity diagram starts by parallelly creating ProductPost and creating an initial draft contract only with the senders' information in it. The system updates it in the news feed so that all other users may see it.

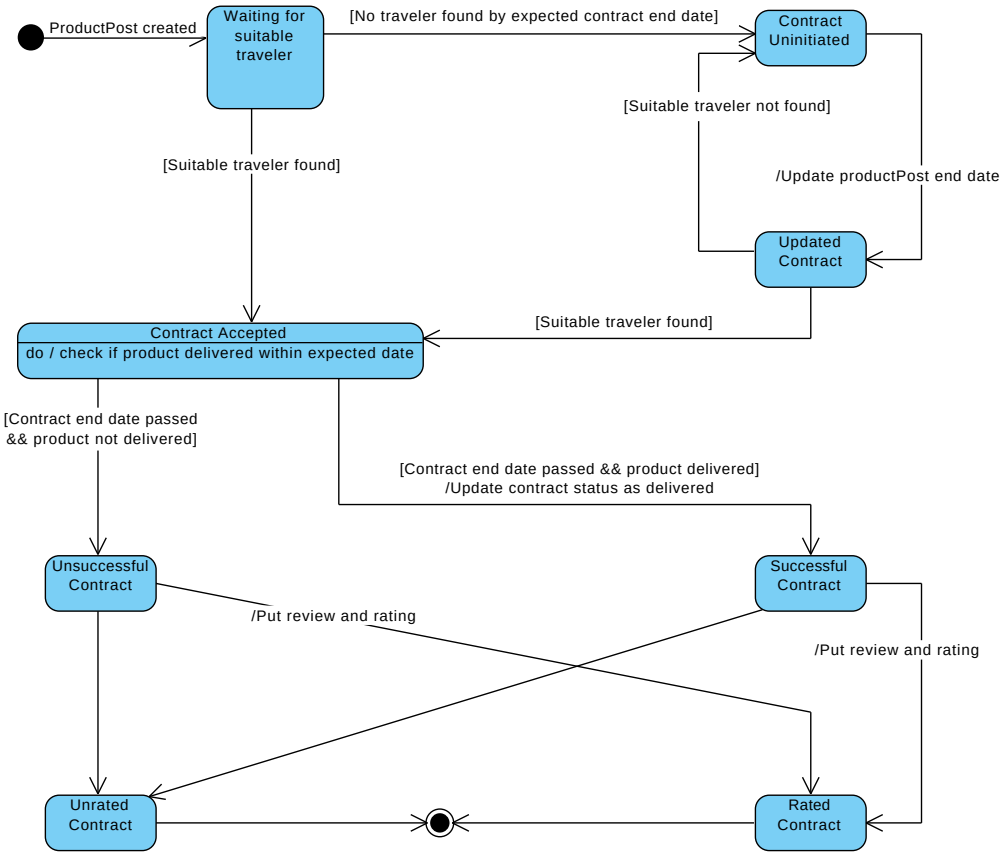


5.2.3 Activity diagram: Reviewing productPost

The activity diagram starts by complaining against an inappropriate ProductPost by any user and the admin reviews it further to decide what to do with this ProductPost.



5.3 State Diagram



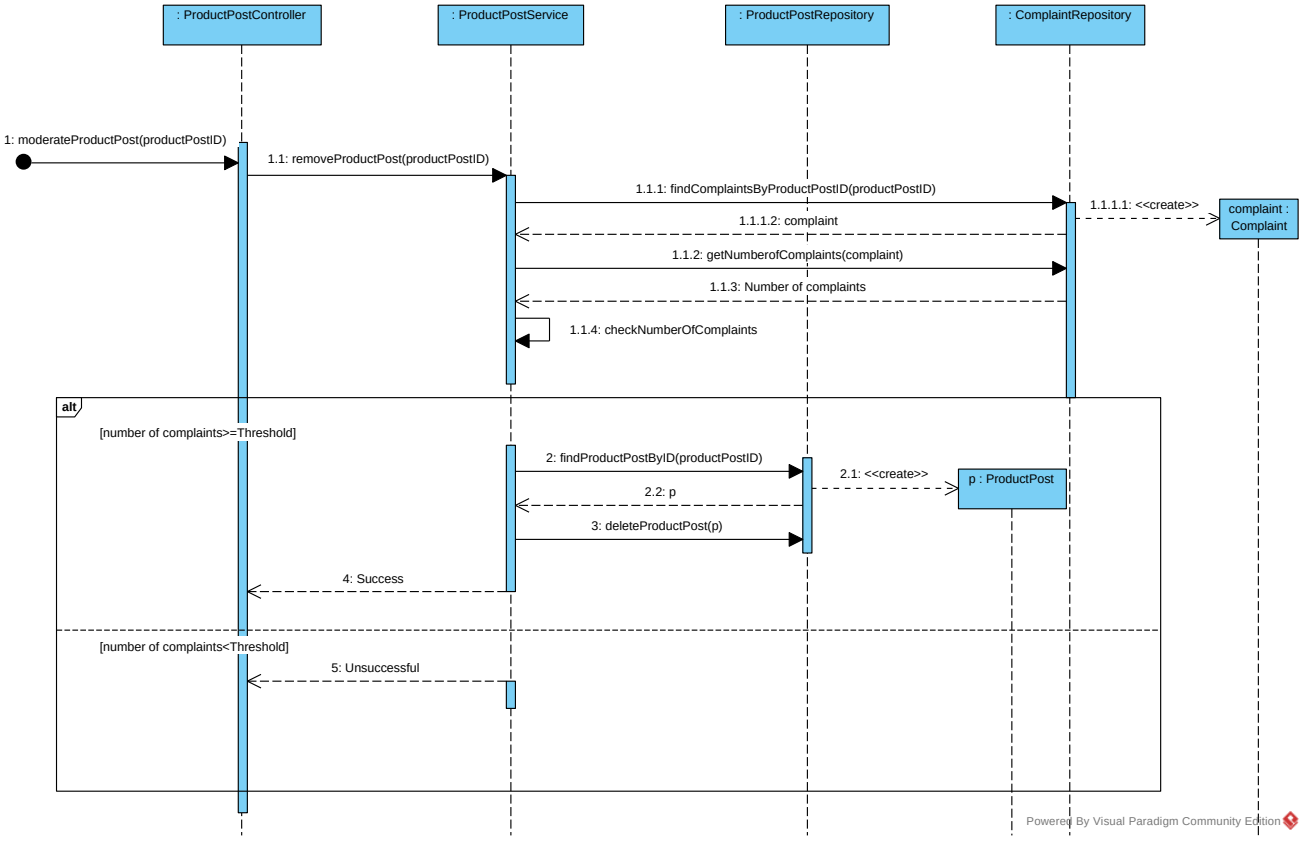
We changed the followings from our previous state machine:

1. We have changed our previous diagram and focused mainly with the state of contract in different stages.
2. After the ProductPost is created and contract is started. So if a suitable traveler is found, contract is updated and waiting for product delivery status. After the date has passed, contract can be updated with rating from user.

5.4 Sequence Diagram

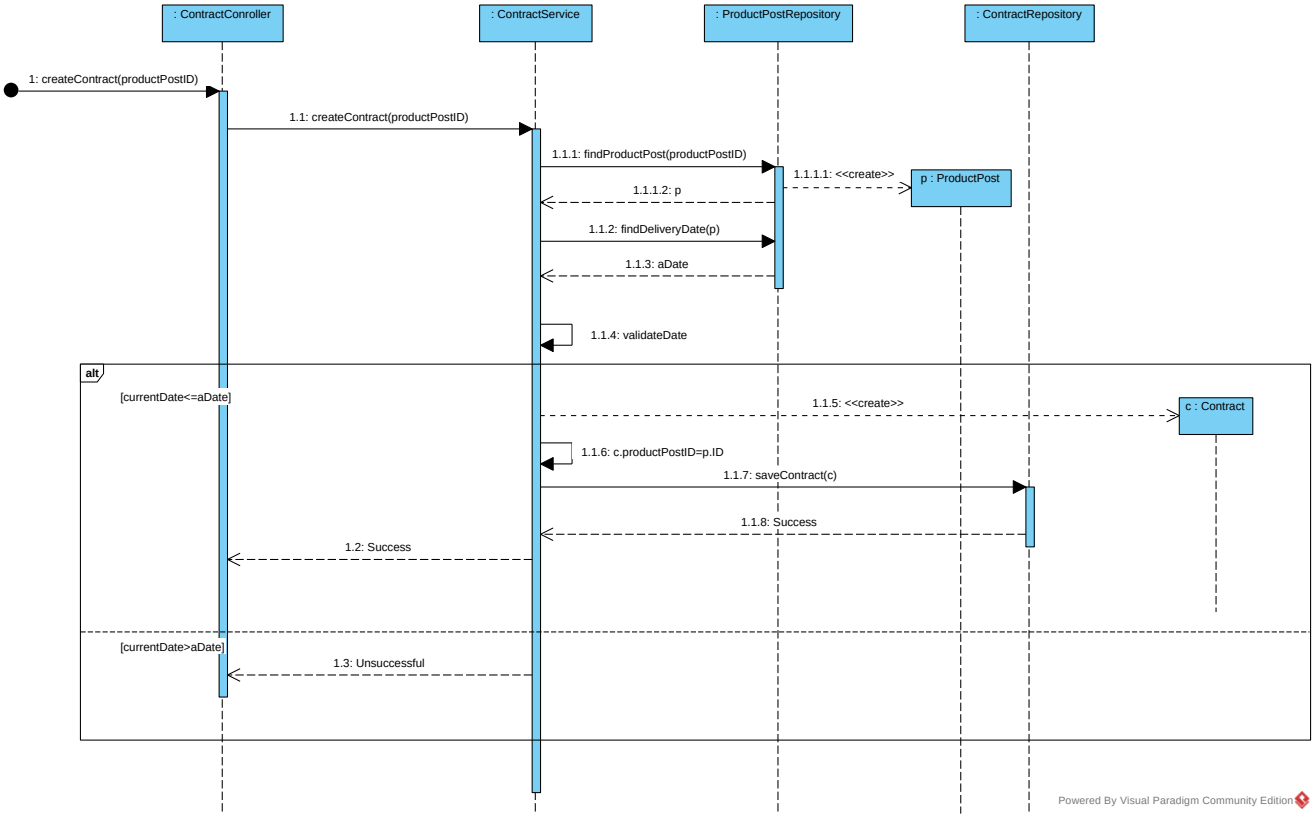
5.4.1 Operation Contract: moderateProductPost.

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



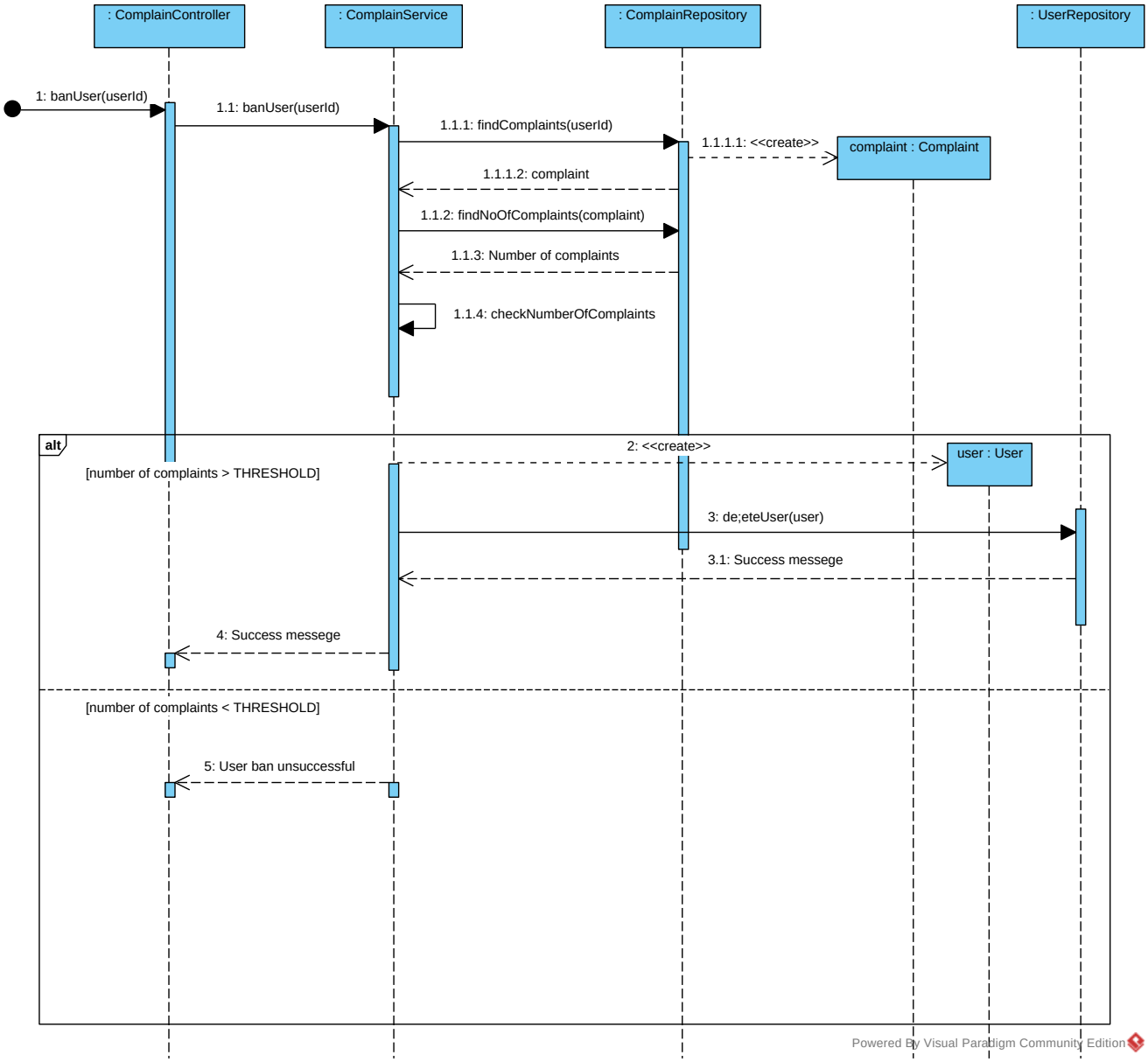
5.4.2 Operation Contract: createContract.

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



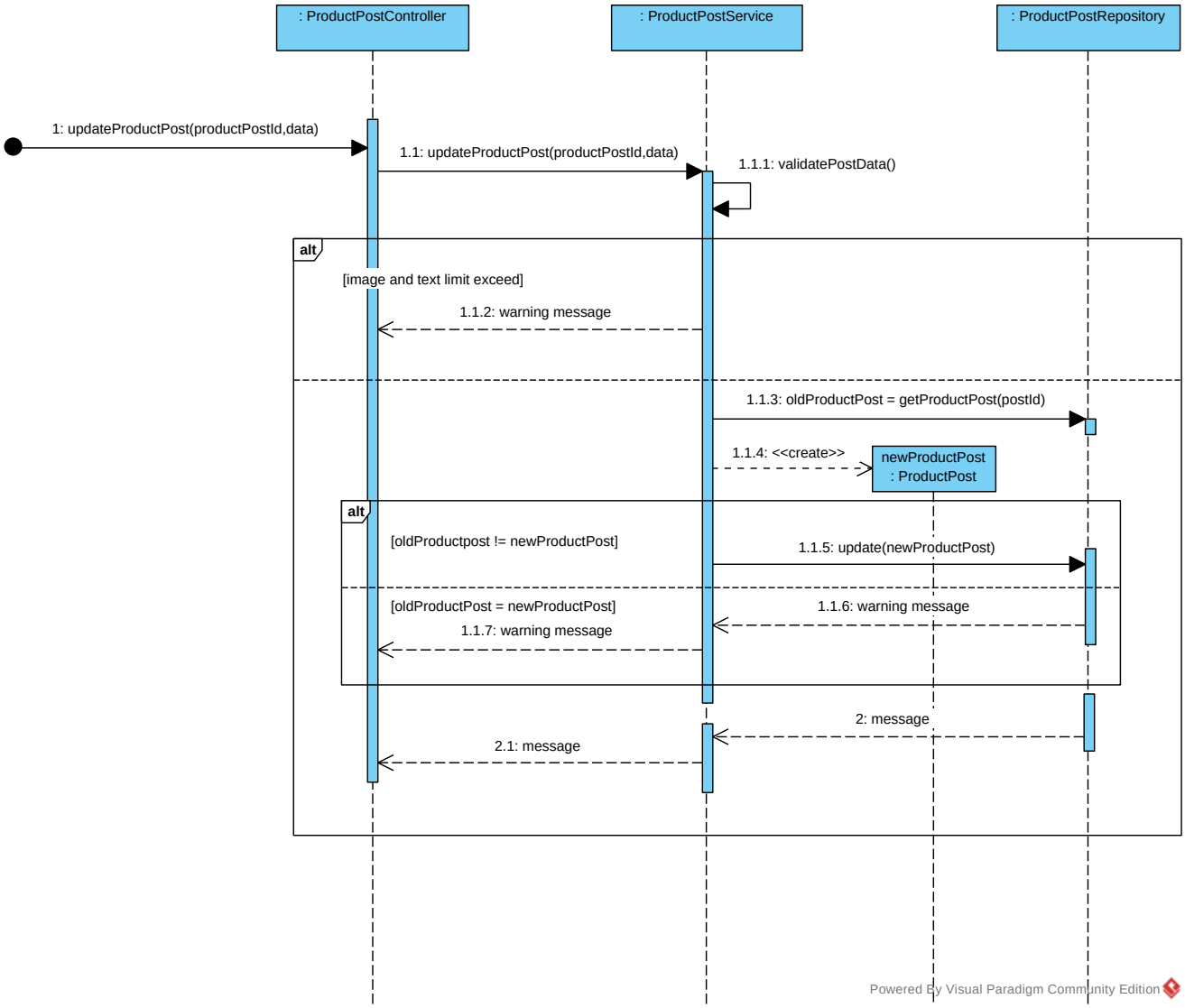
5.4.3 Operation Contract: banUser.

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



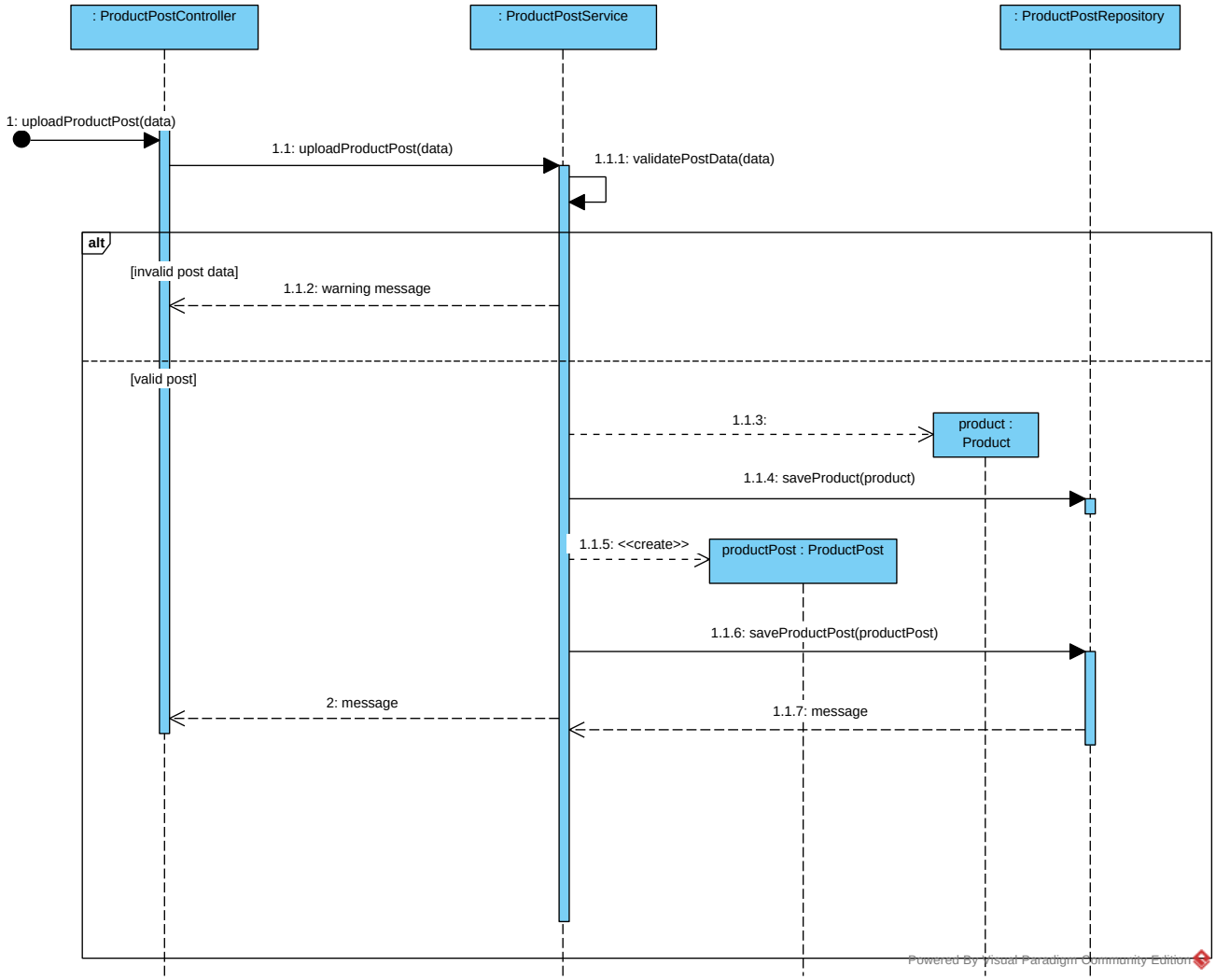
5.4.4 Operation Contract: updateProductPost.

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



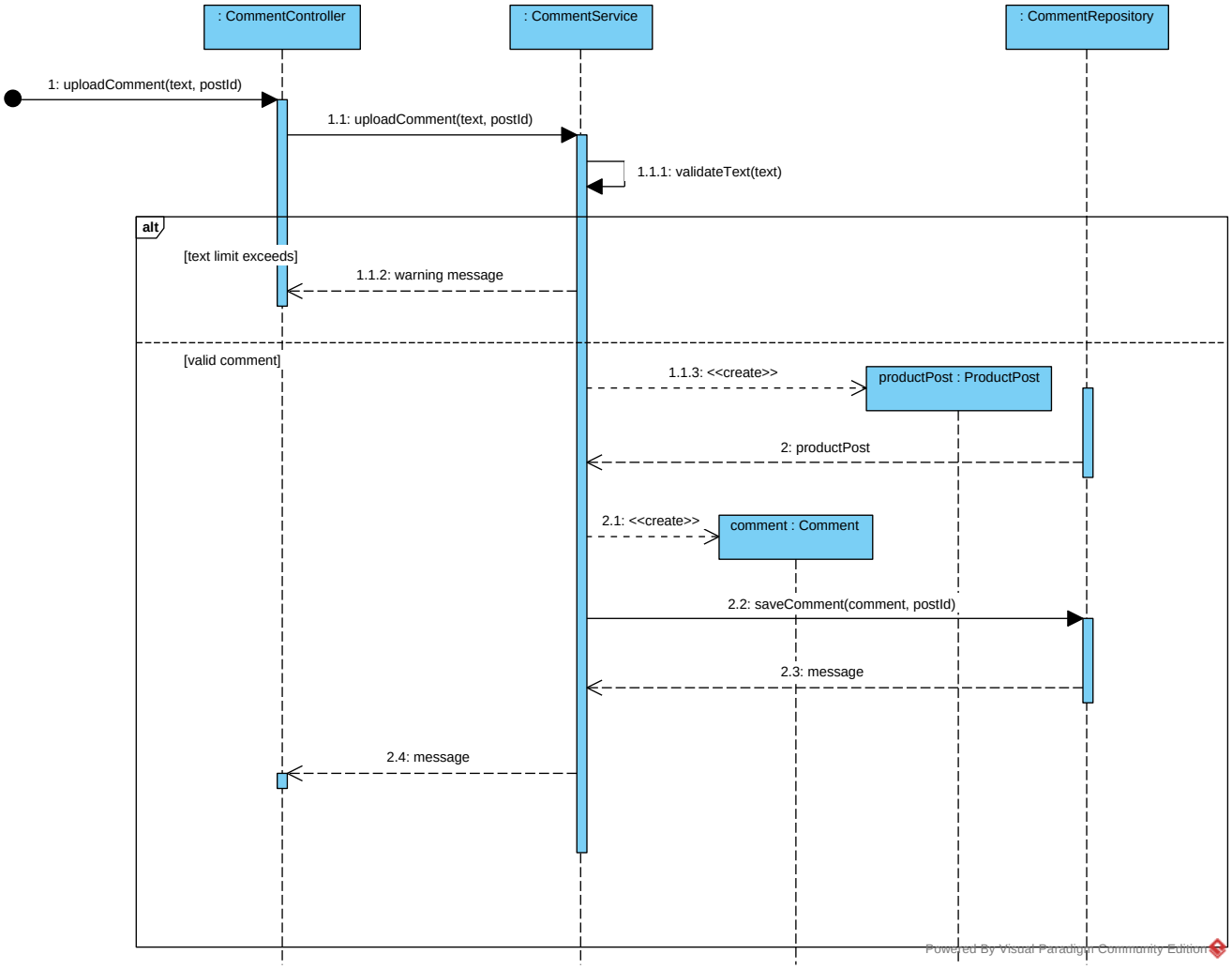
5.4.5 Operation Contract: uploadProductPost.

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



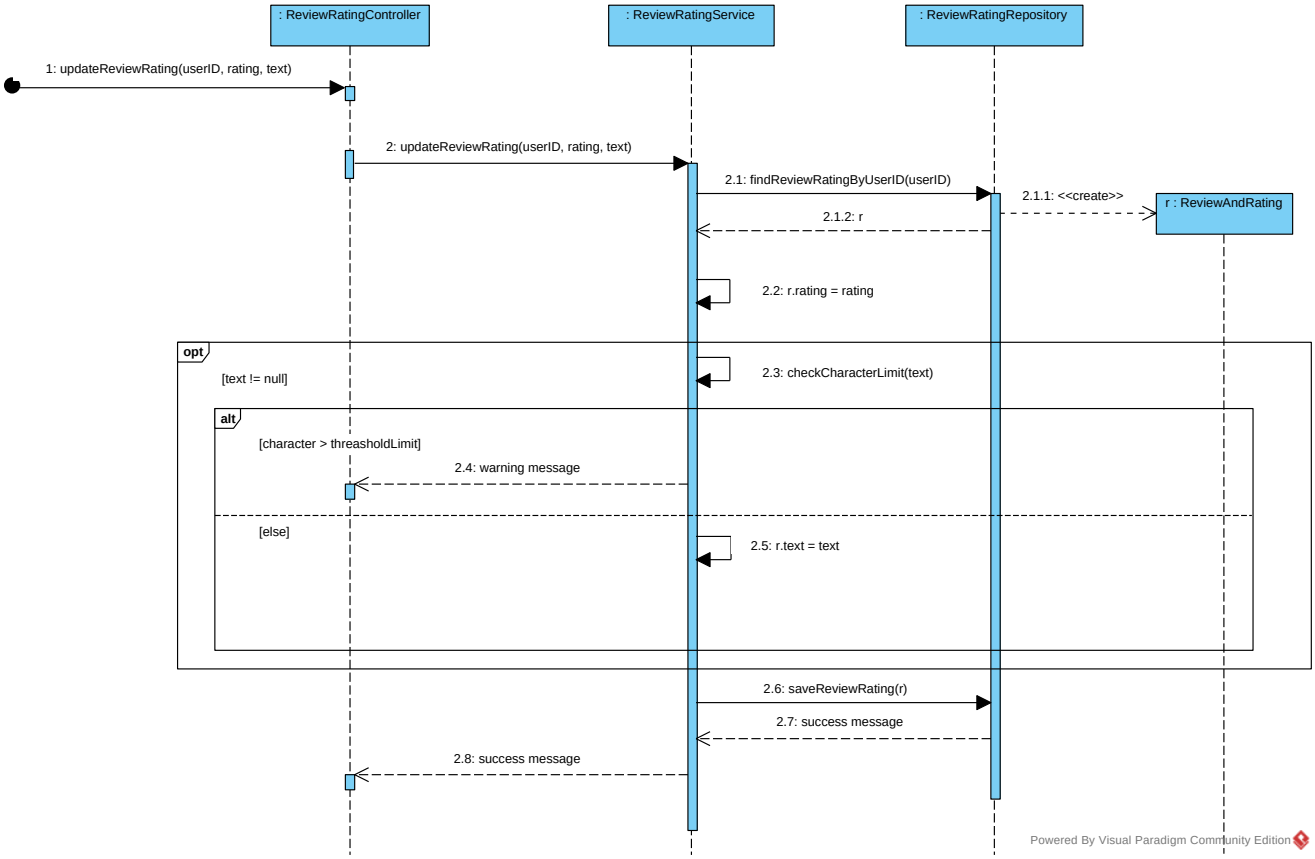
5.4.6 Operation Contract: uploadComment.

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



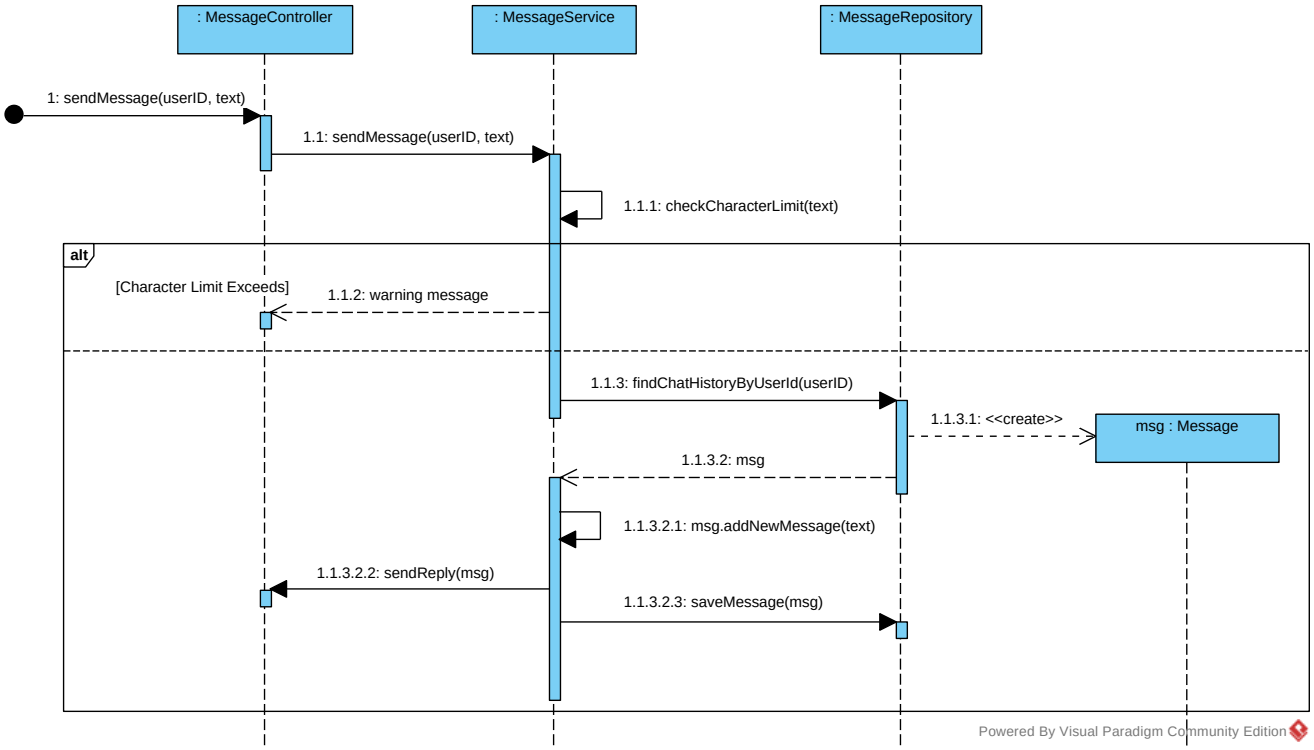
5.4.7 Operation Contract: updateReviewRating.

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



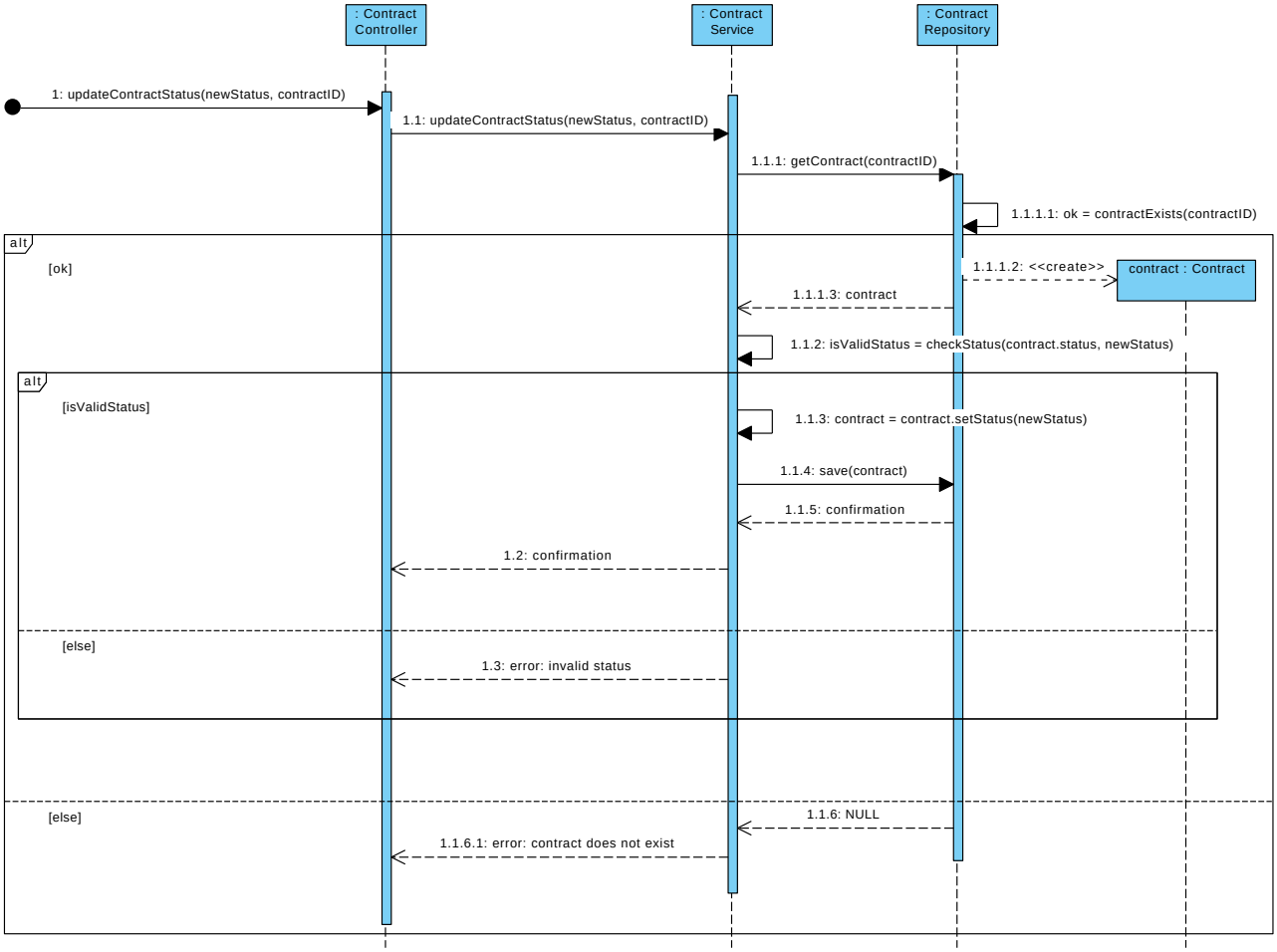
5.4.8 Operation Contract: sendMessage.

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



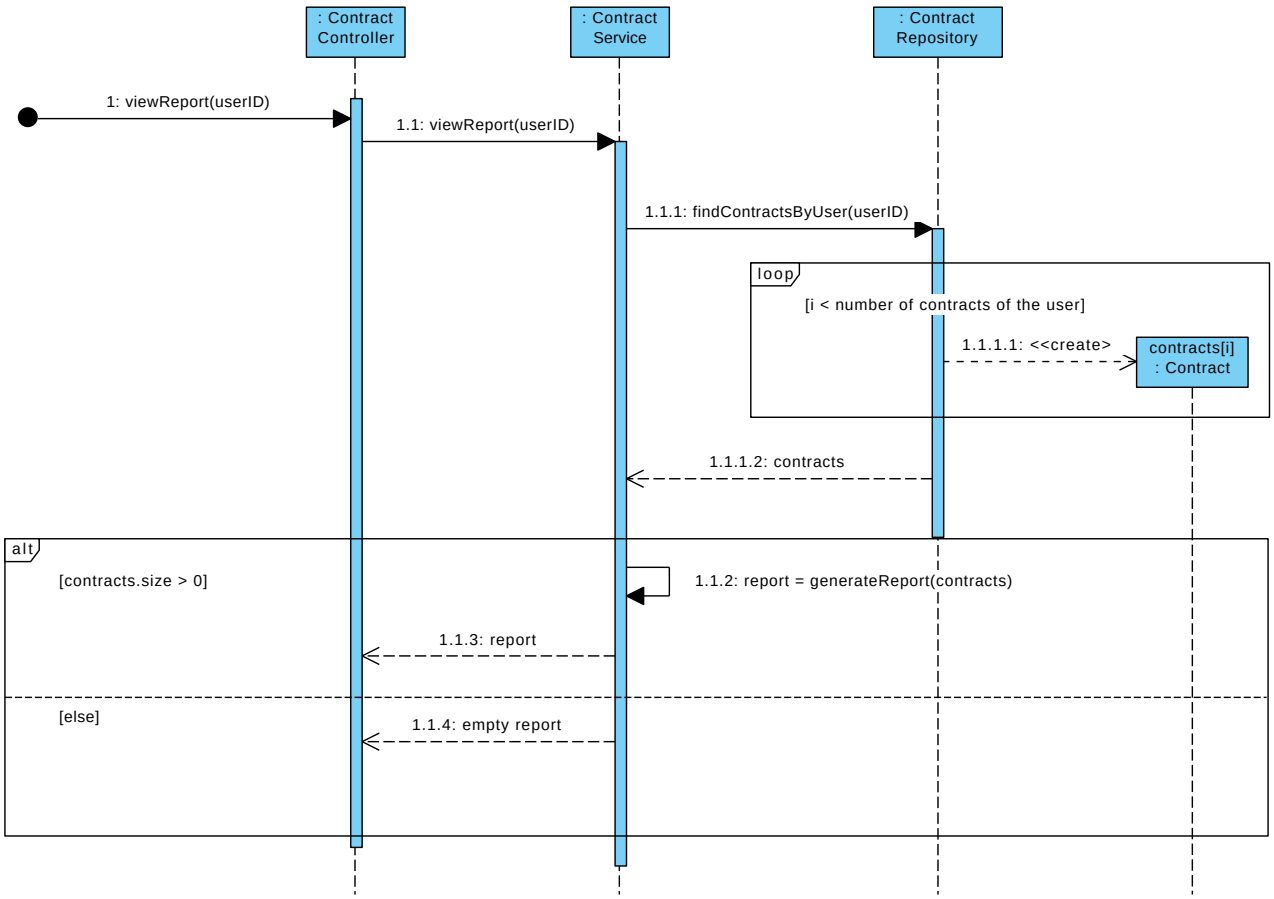
5.4.9 Operation Contract: updateContractStatus.

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



5.4.10 Operation Contract: viewReport.

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



5.5 Object Constraint Language (OCL)

5.5.1 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
```

5.5.2 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
```

5.5.3 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
```

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

5.5.4 Admin constraint

Constraint: There must be at least one administrator.

```
context User

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

5.5.5 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)
```

5.5.6 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)
```

5.5.7 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)
```

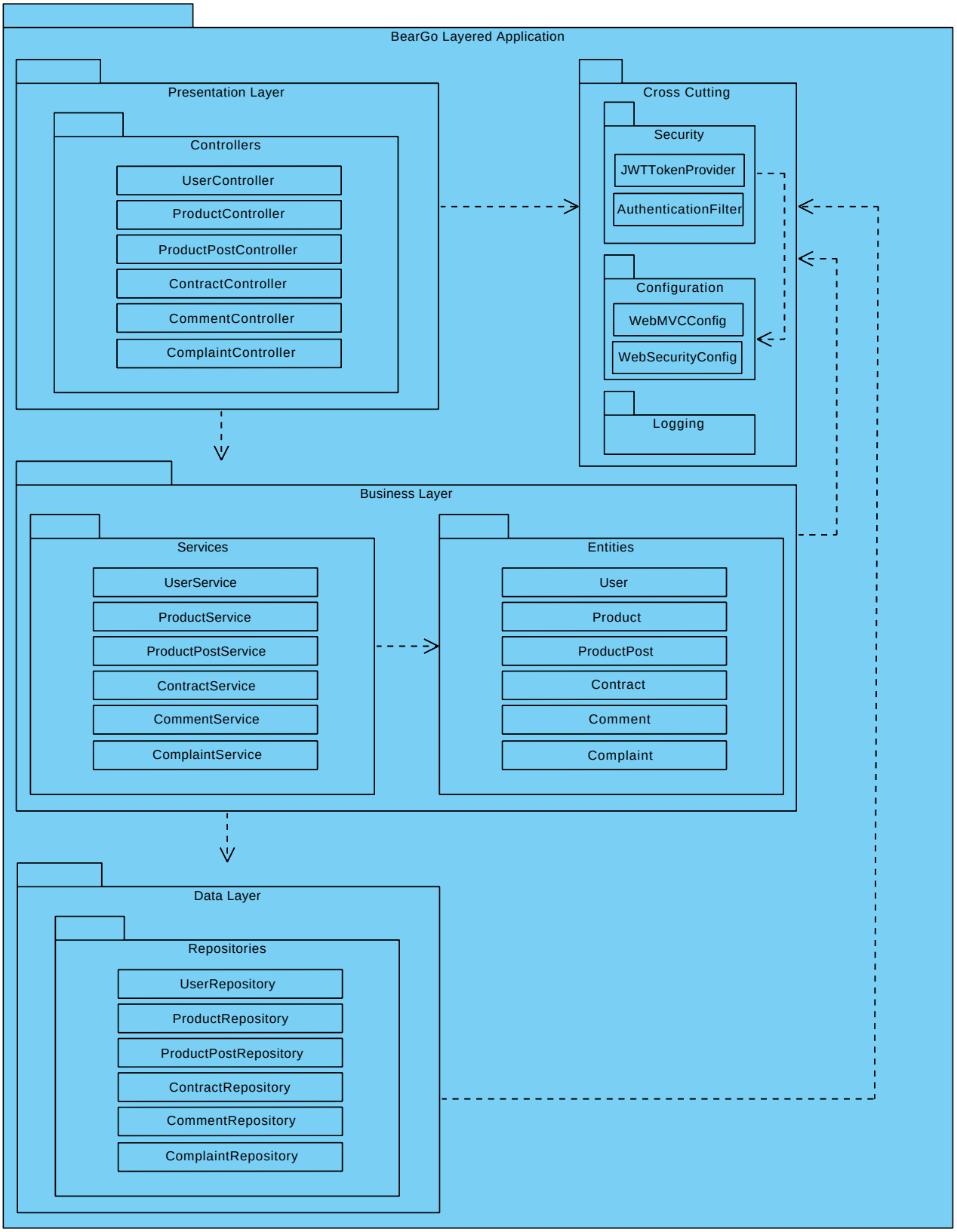
5.5.8 PromoteUser constraint

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

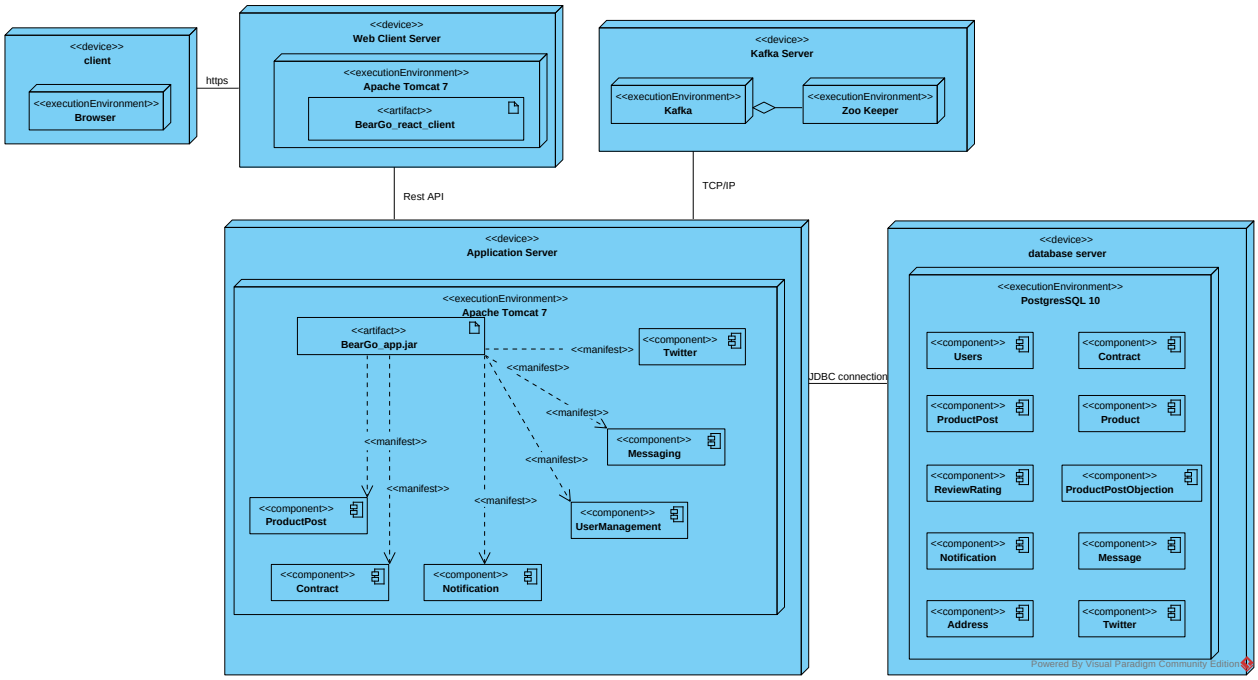
```
context User::promoteUser(t : User)

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

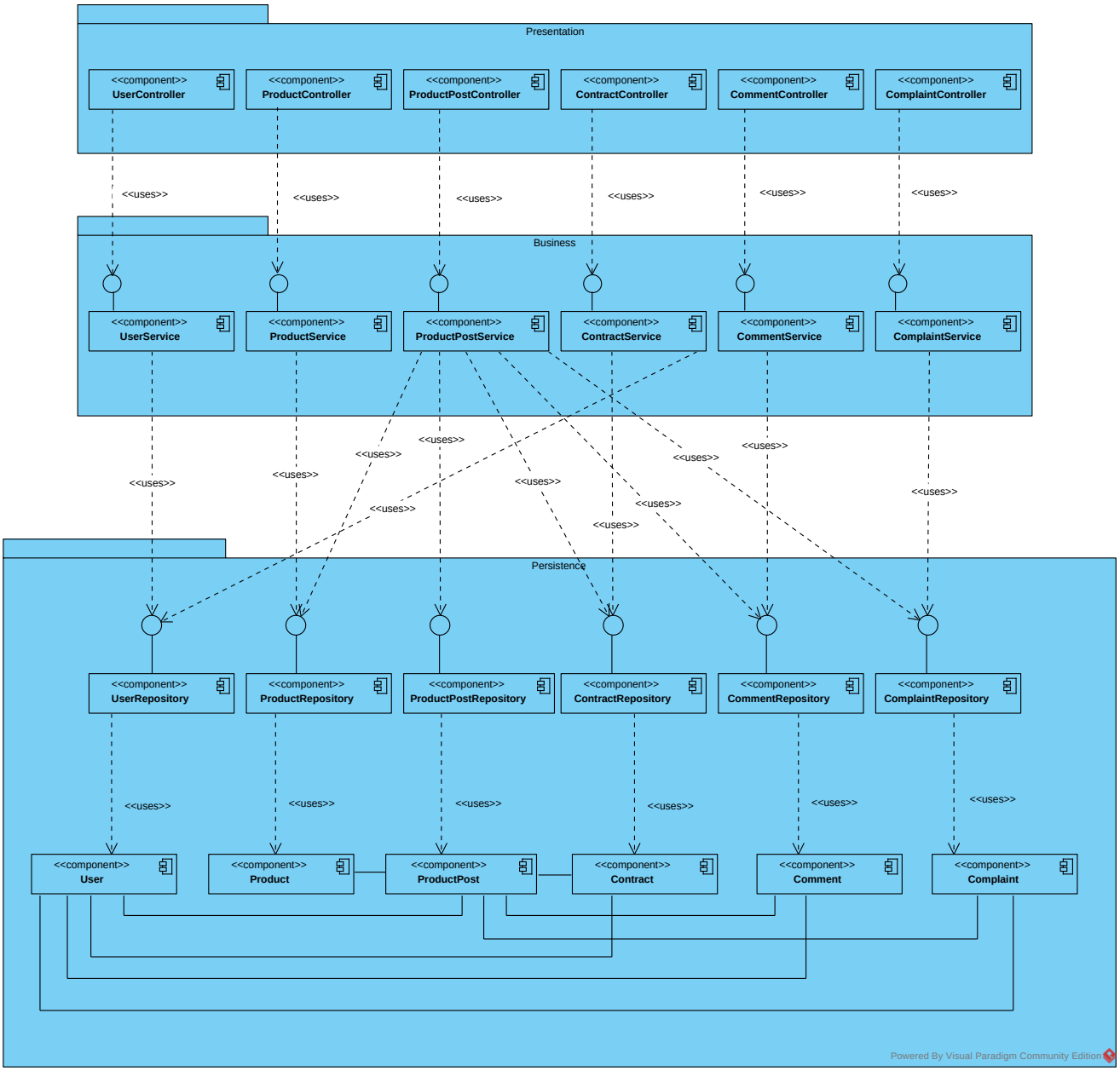
5.6 Package Diagram



5.7 Deployment Diagram

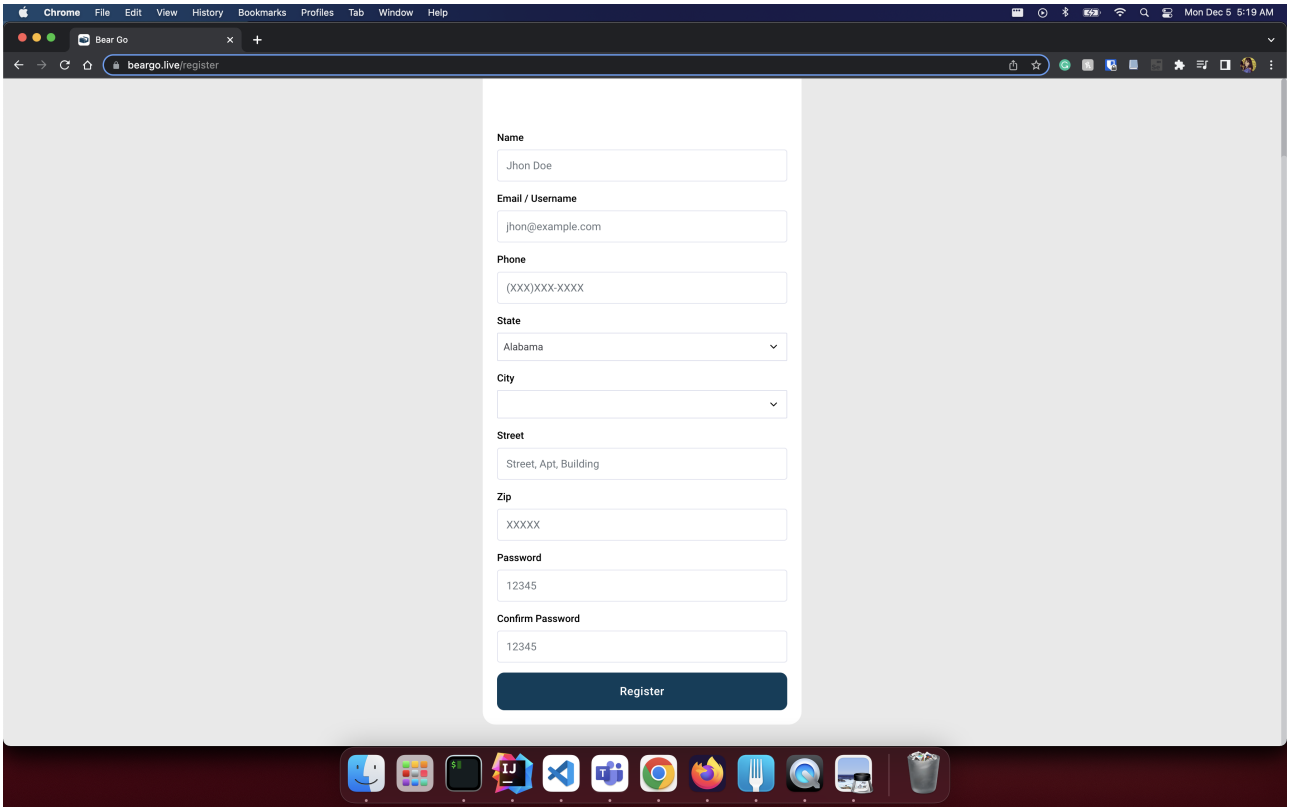


5.8 Component Diagram

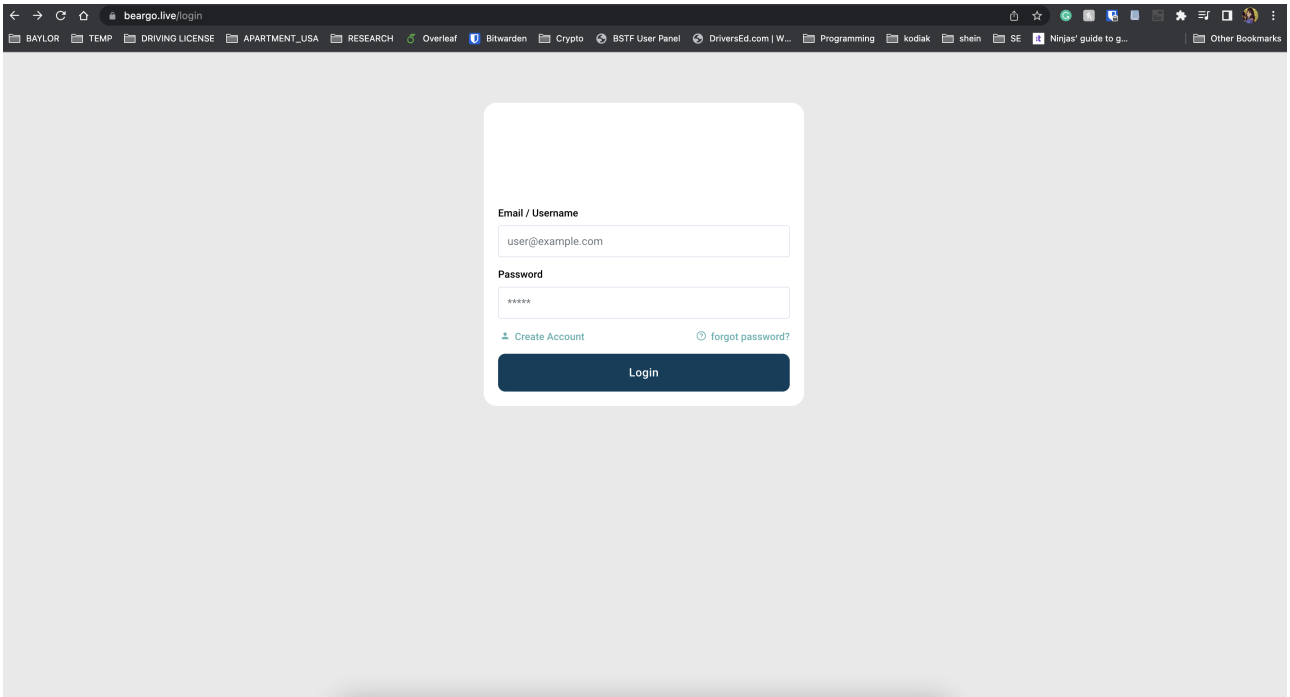


6 UI Overview

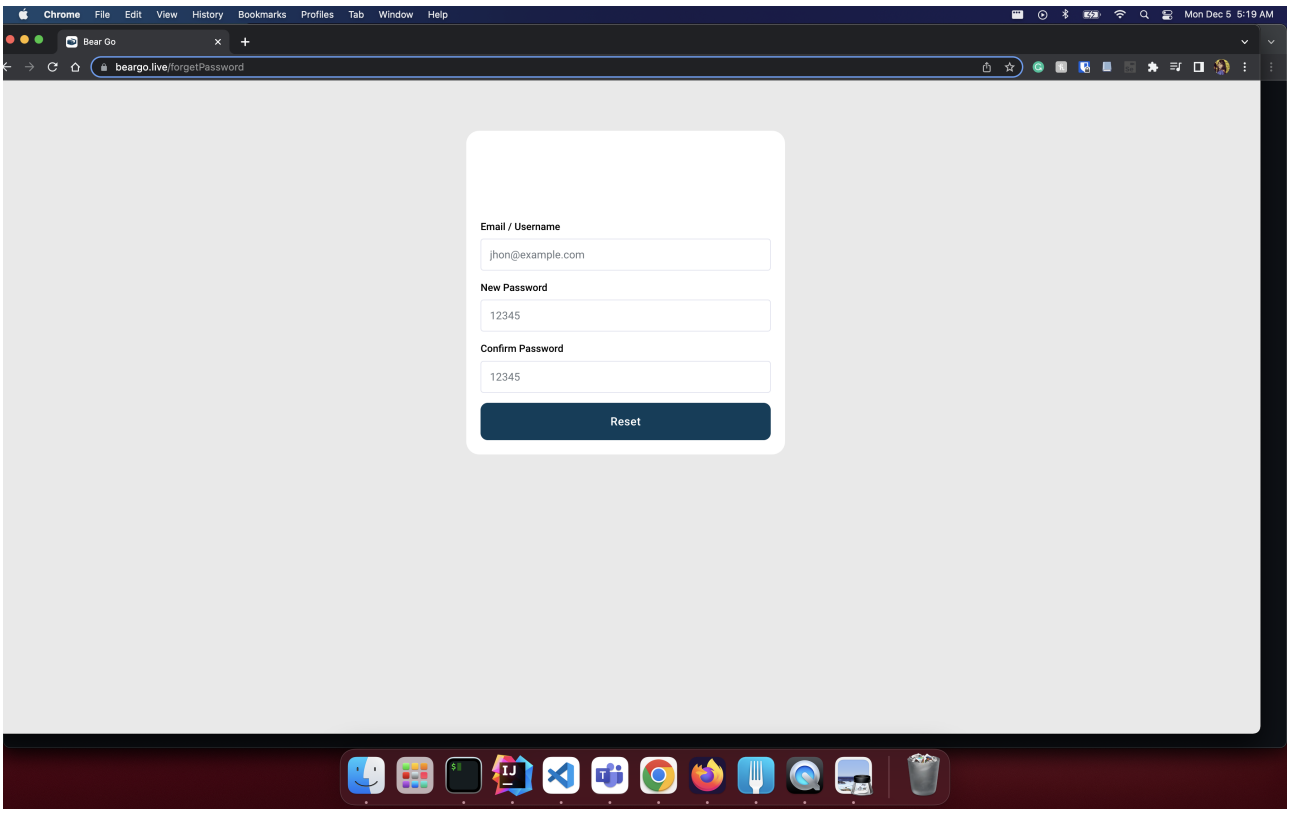
6.1 Registration Page



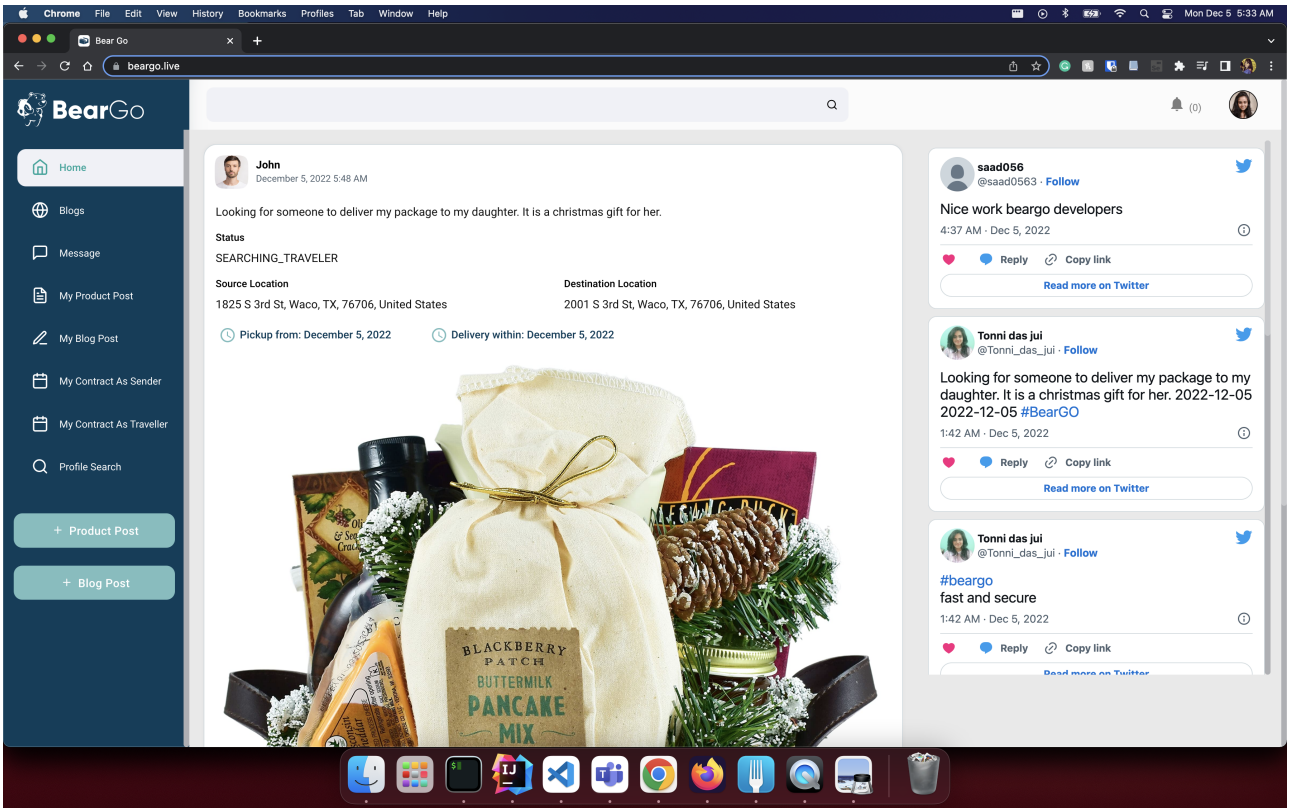
6.2 Log In Page



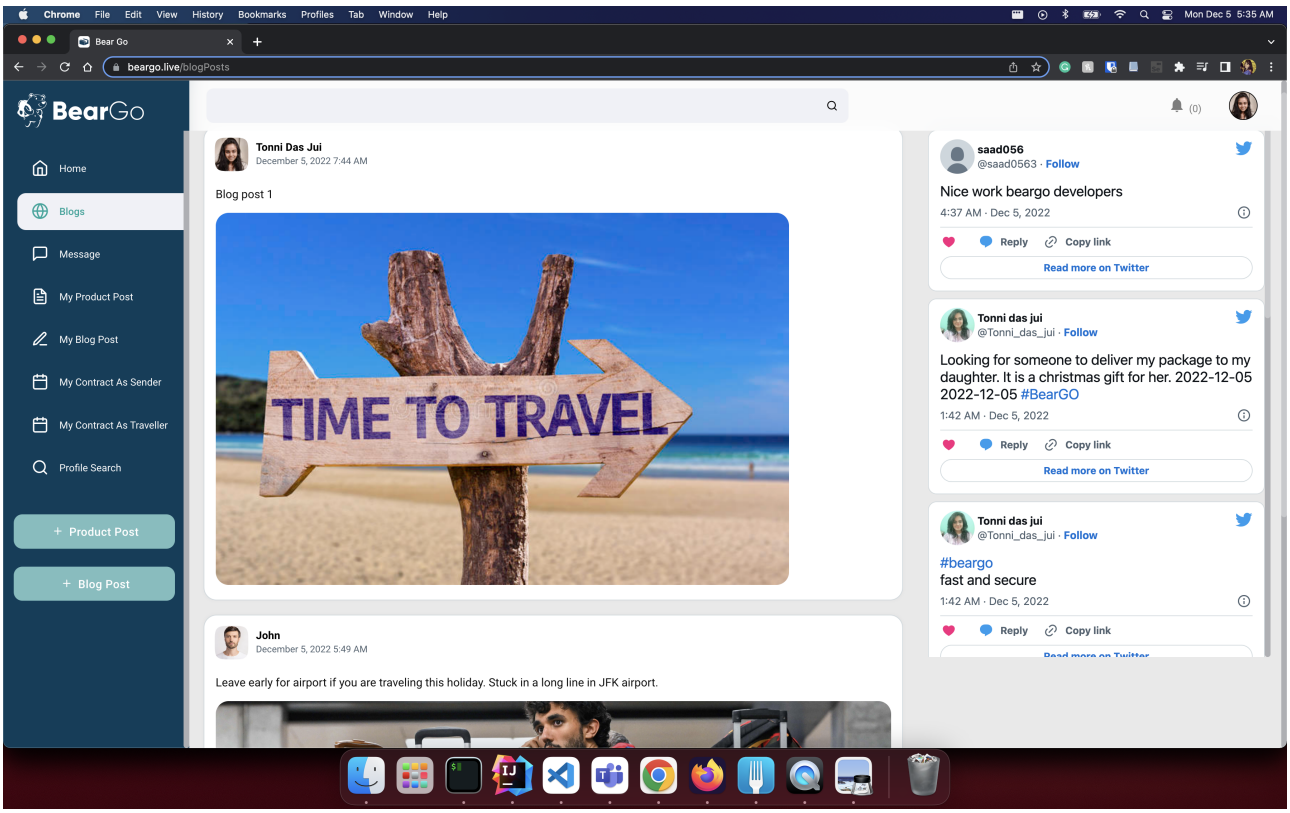
6.3 Forget Password Page



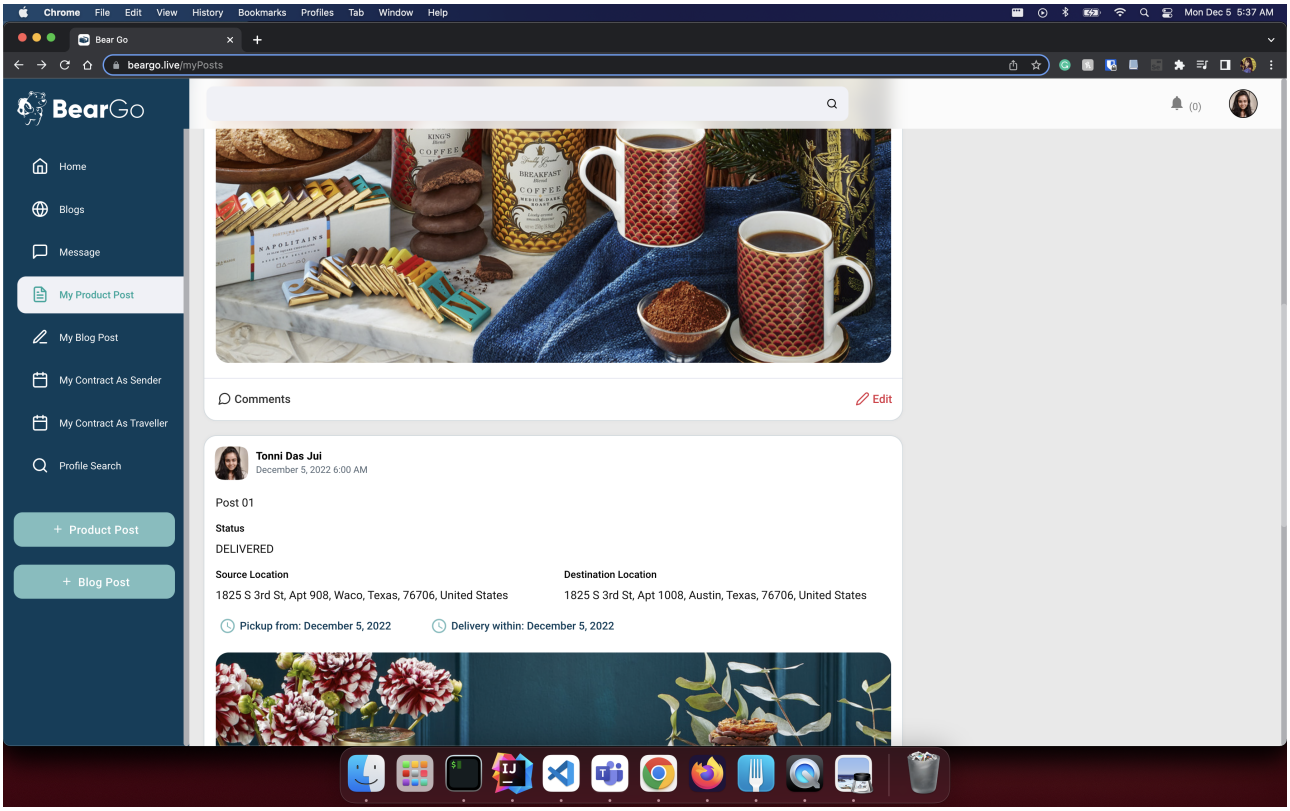
6.4 Home Page



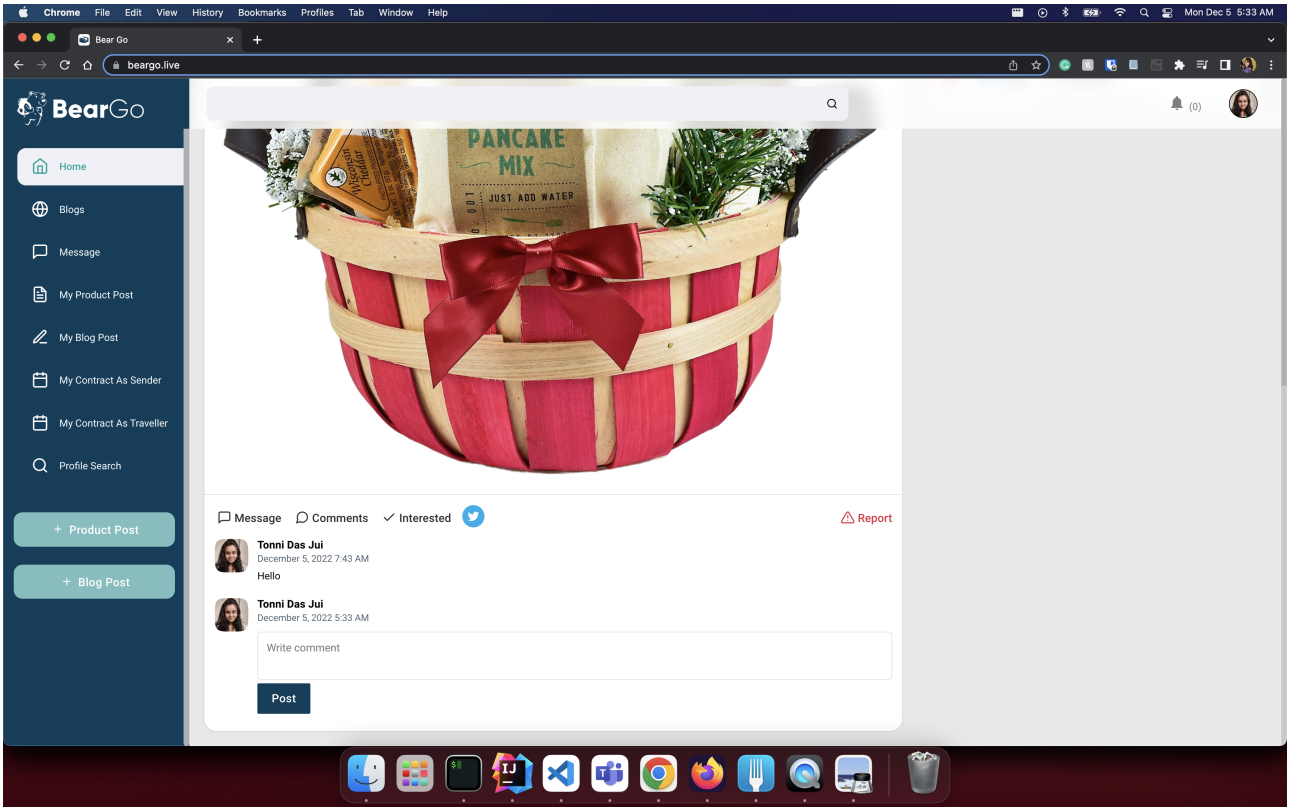
6.5 Blog Page



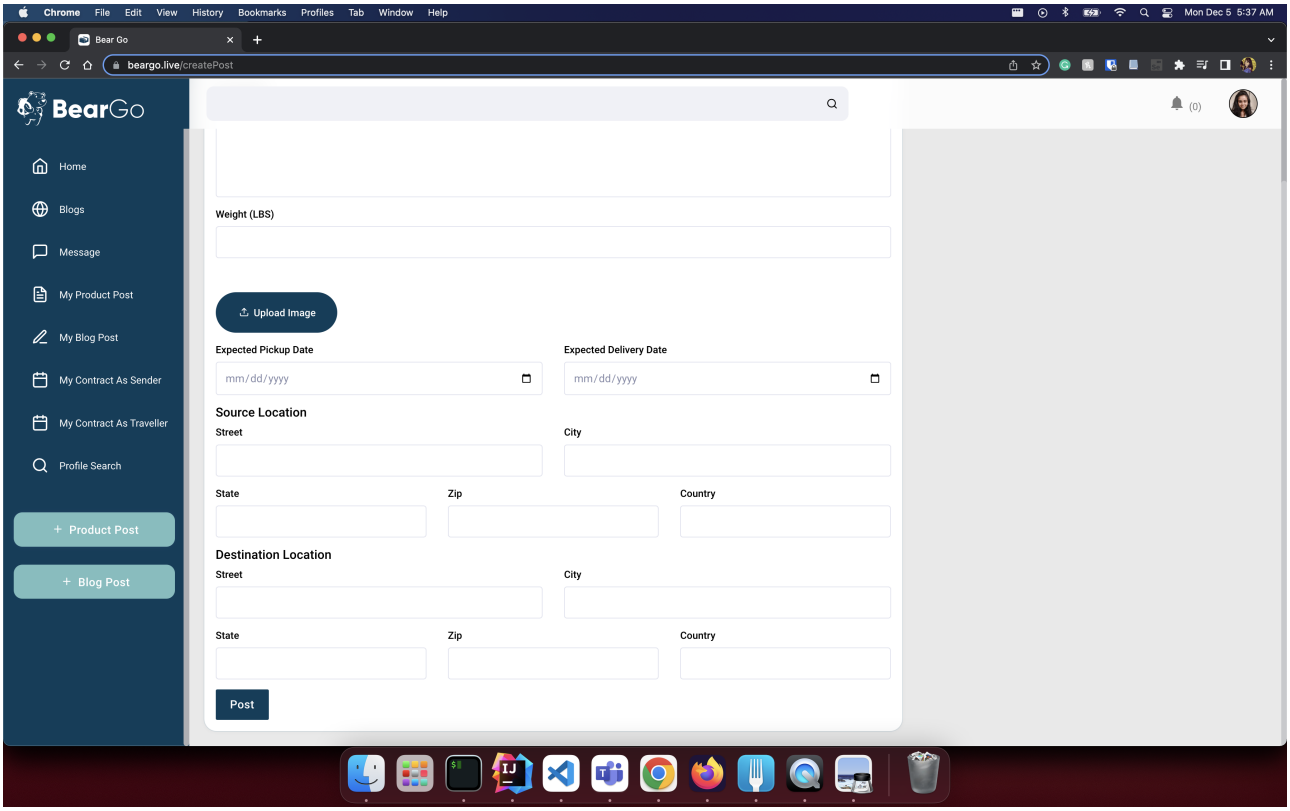
6.6 My ProductPost Page (Edit ProductPost)



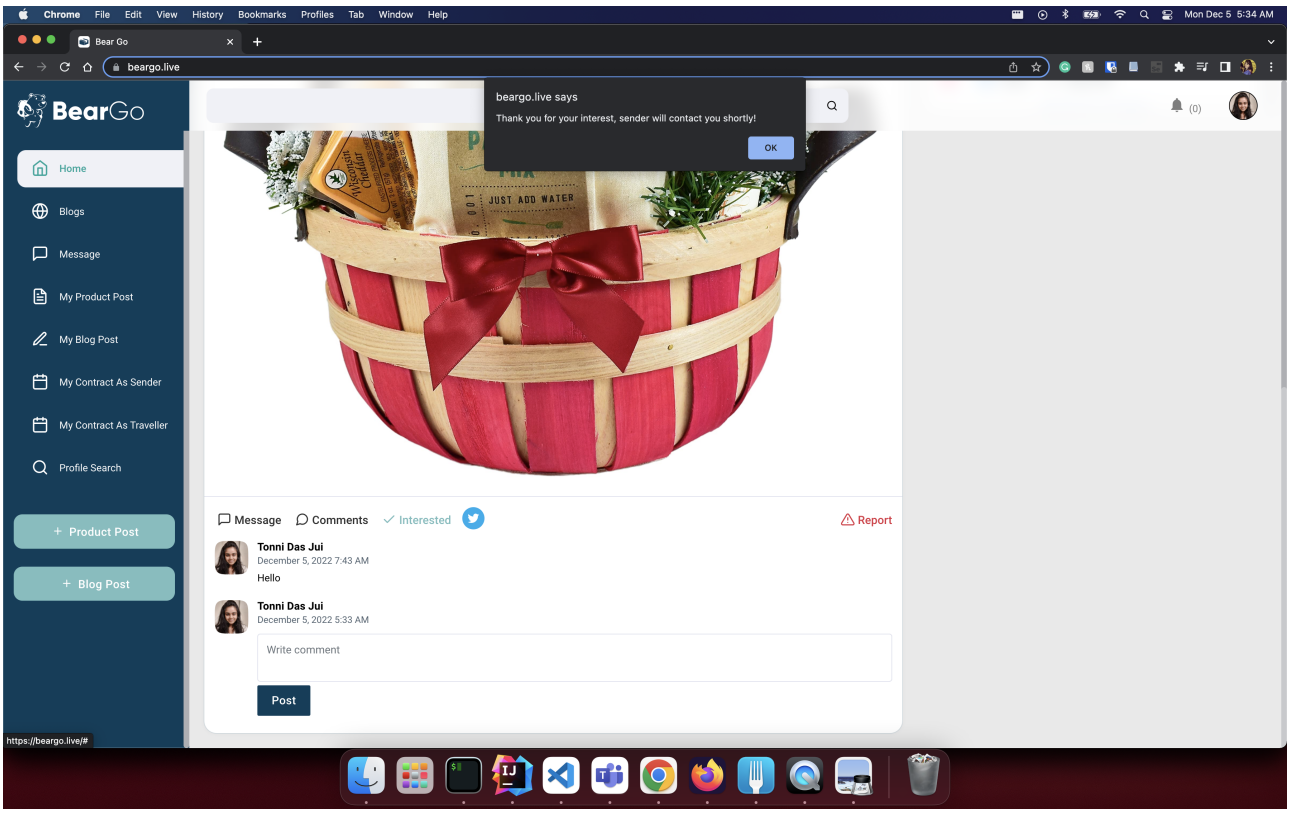
6.7 Comments Section



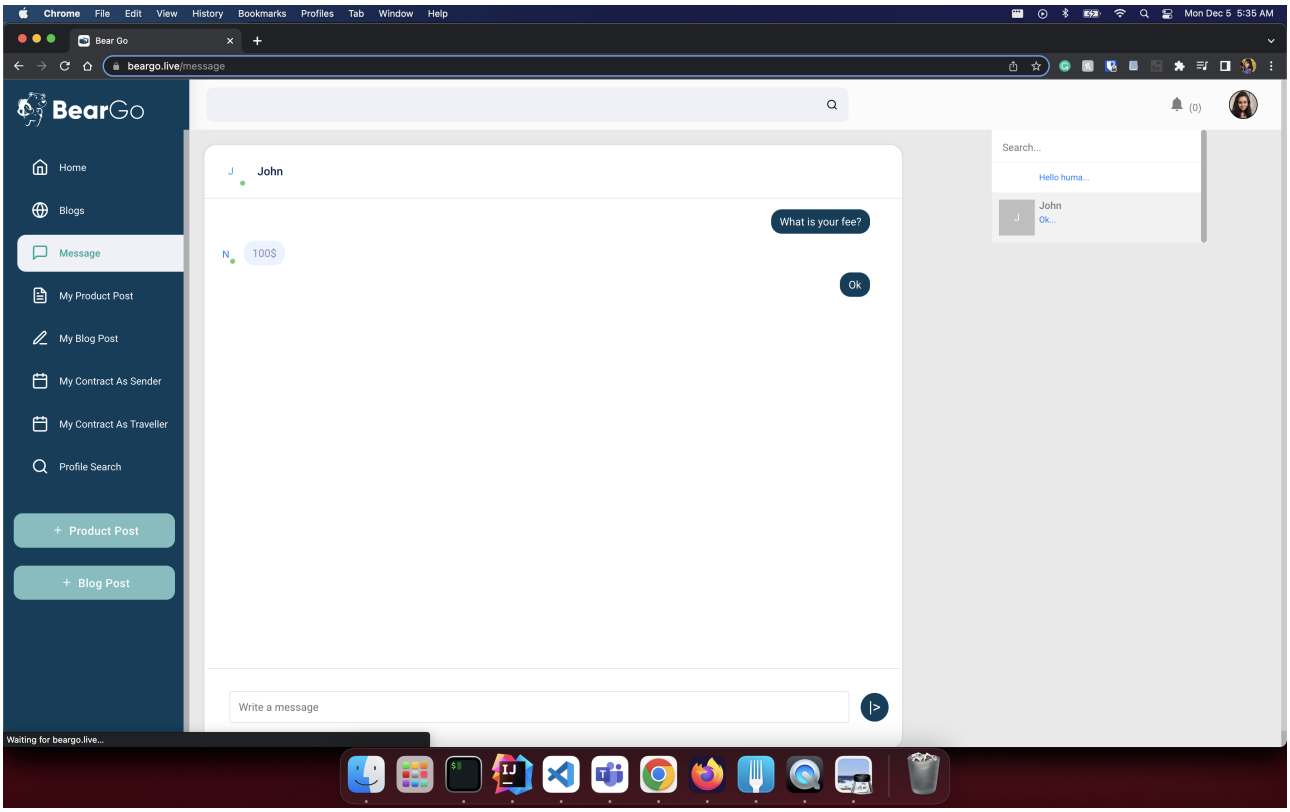
6.8 Create ProductPost Page



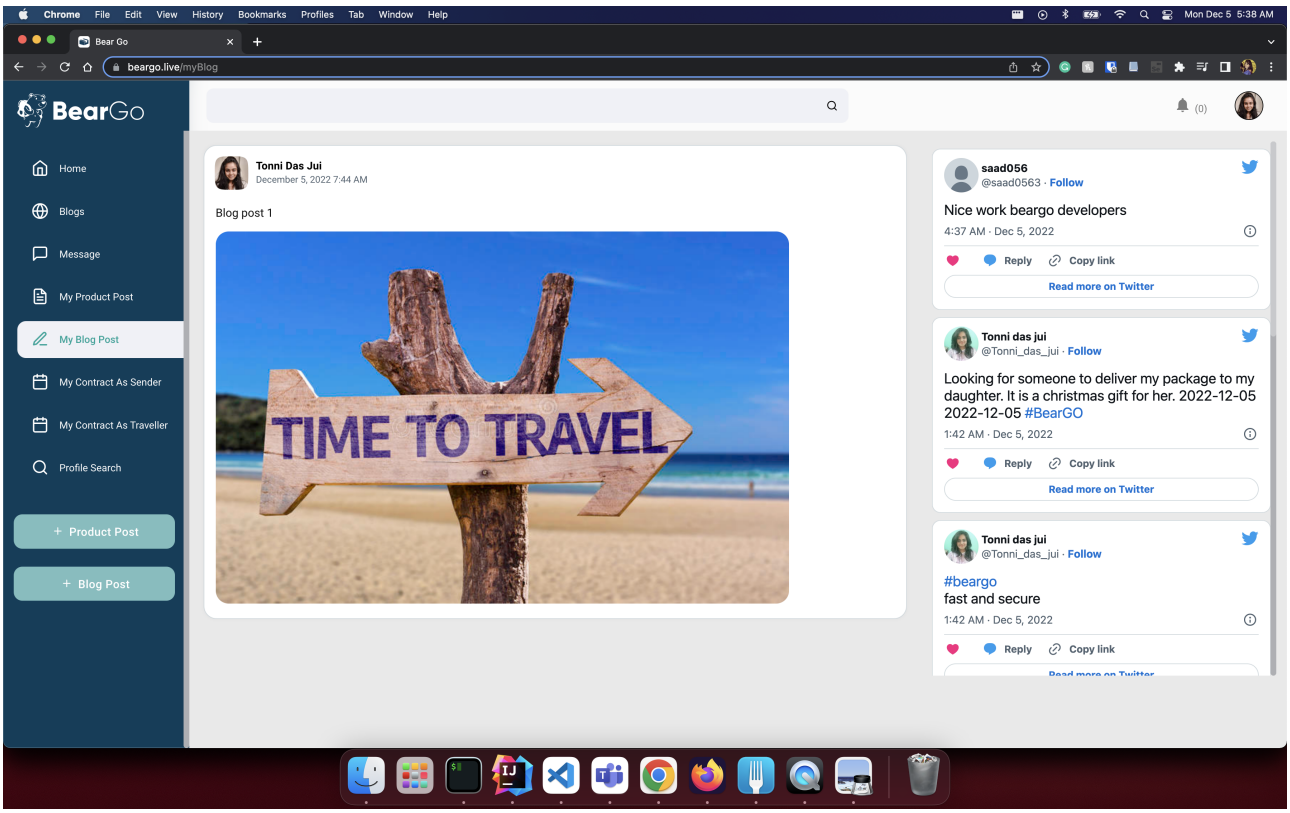
6.9 Traveler request Button



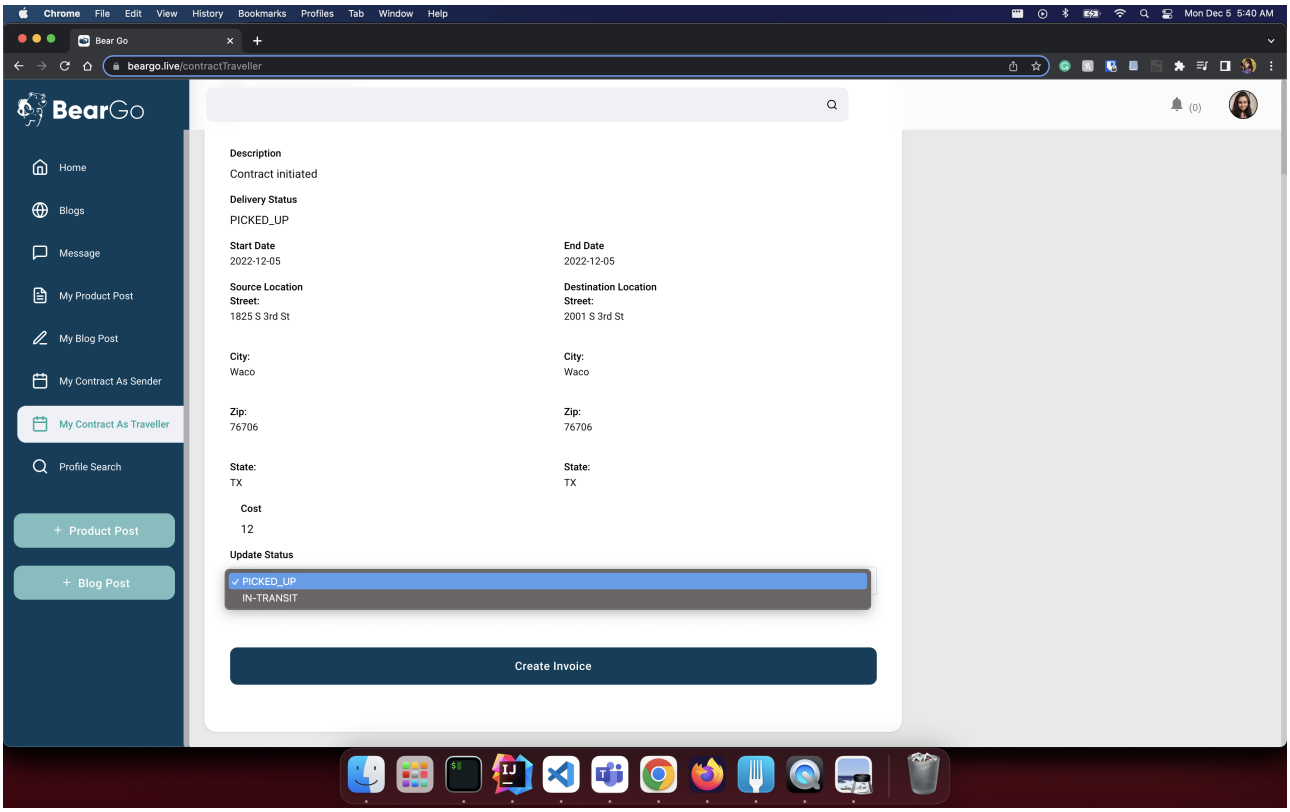
6.10 Message Page



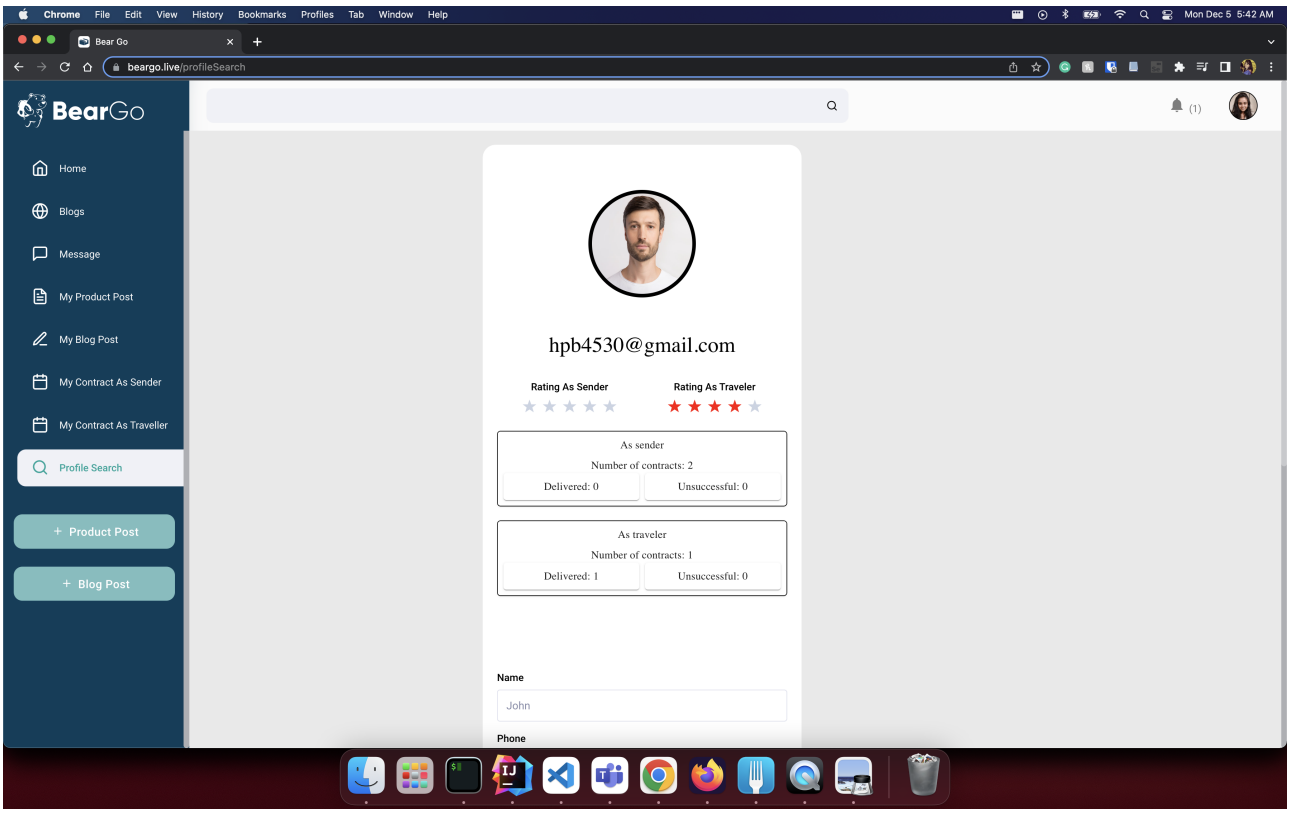
6.11 My Blog Page



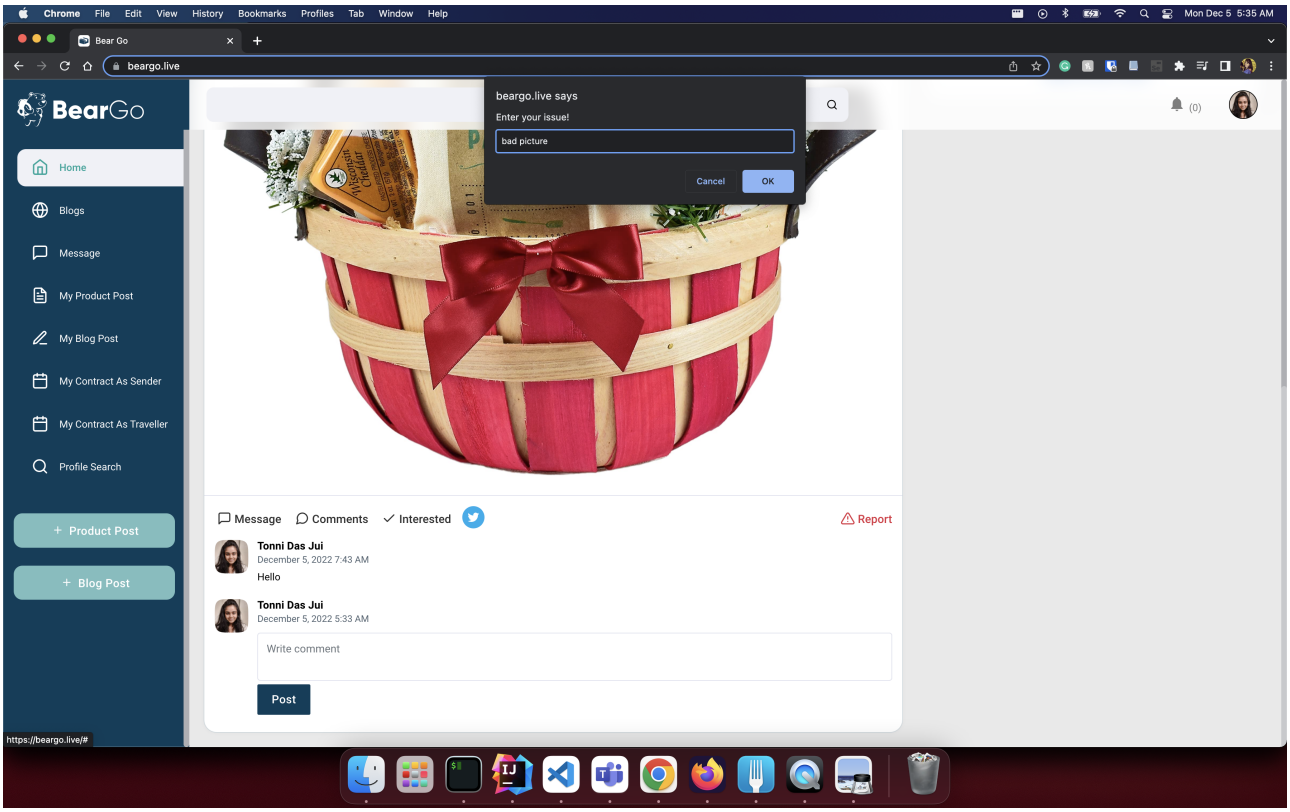
6.12 Product Tracking status change Page



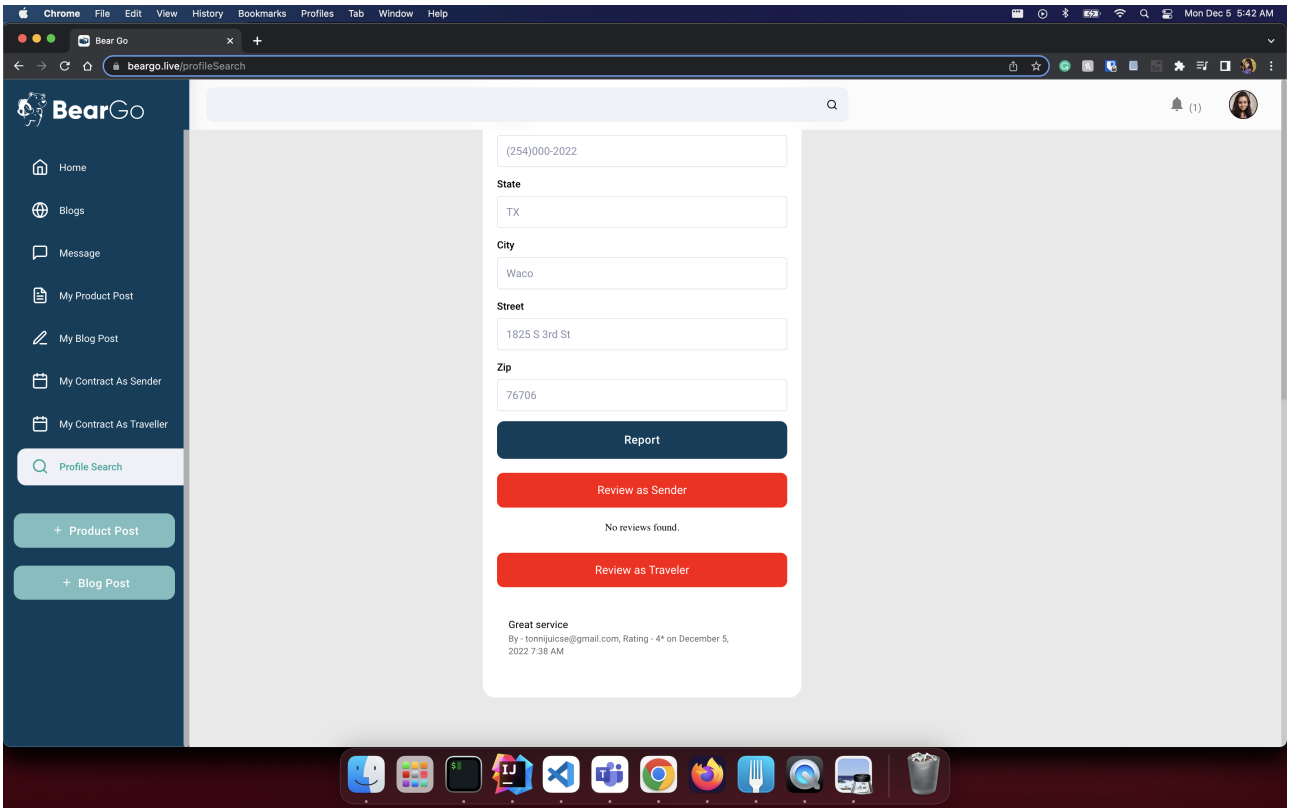
6.13 Other's Profile View Page



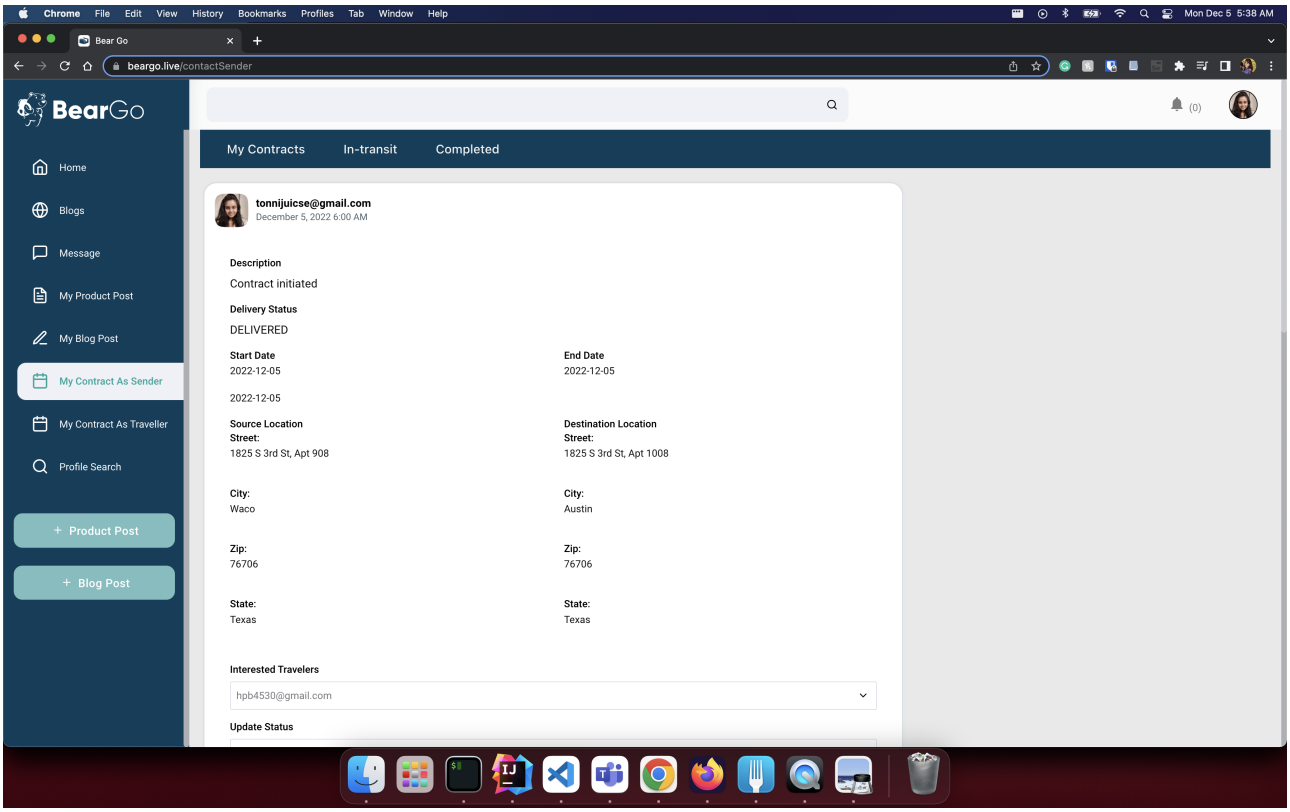
6.14 Complaining against a ProductPost with reason Page



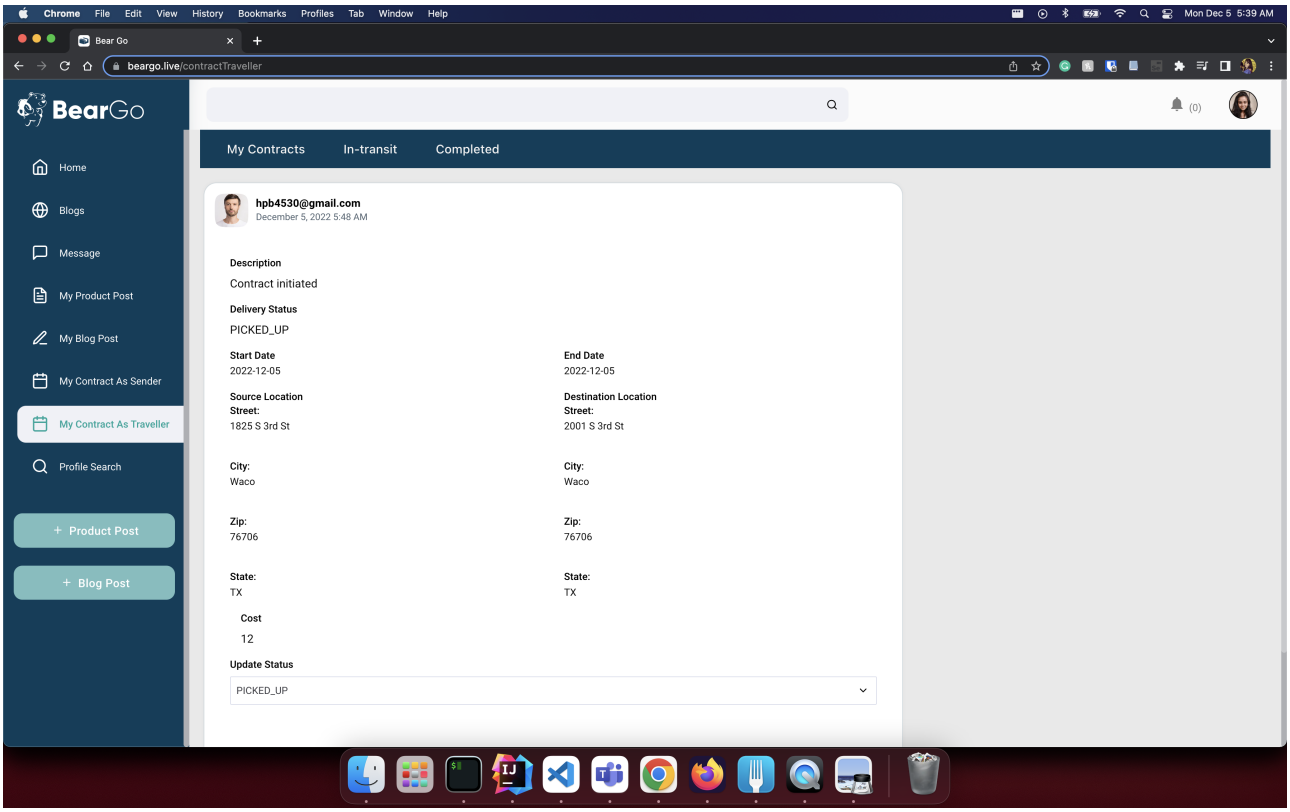
6.15 Review Rating in others Profile



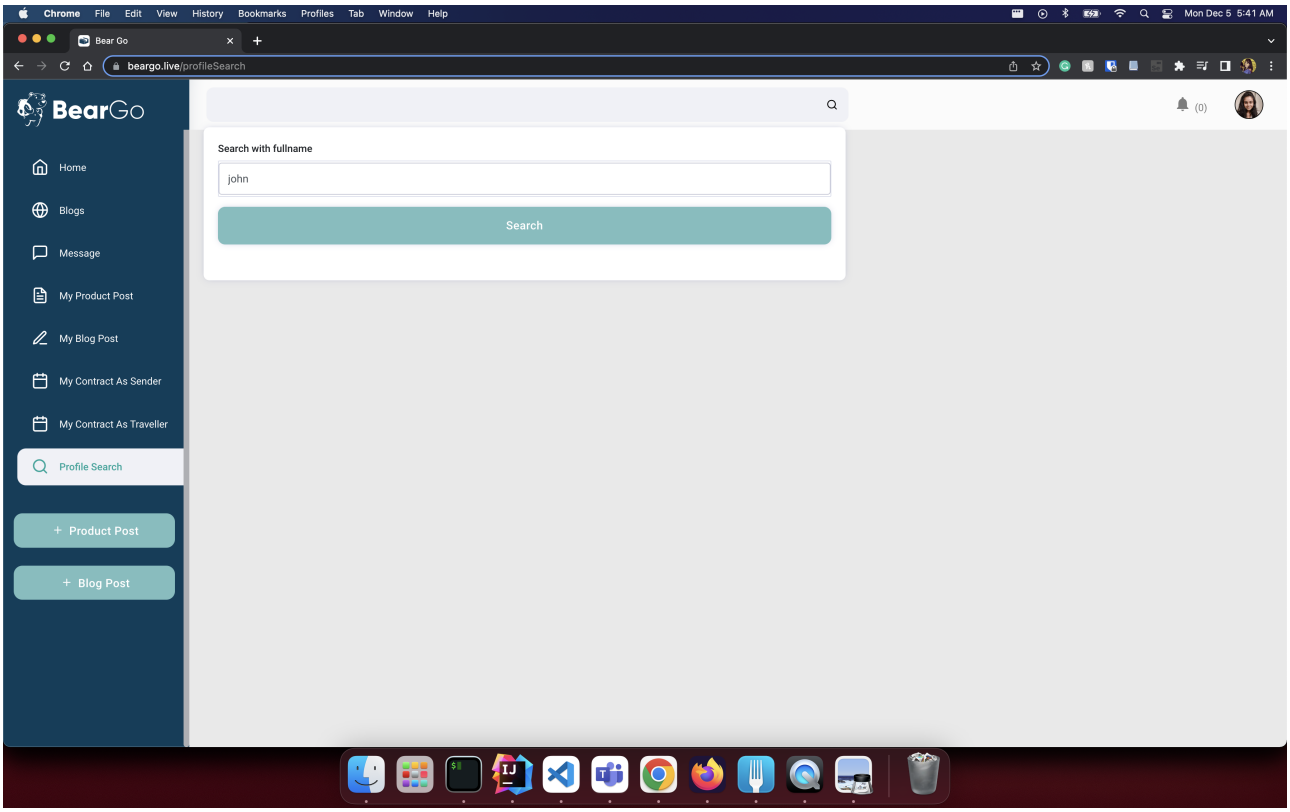
6.16 Sender contract Page



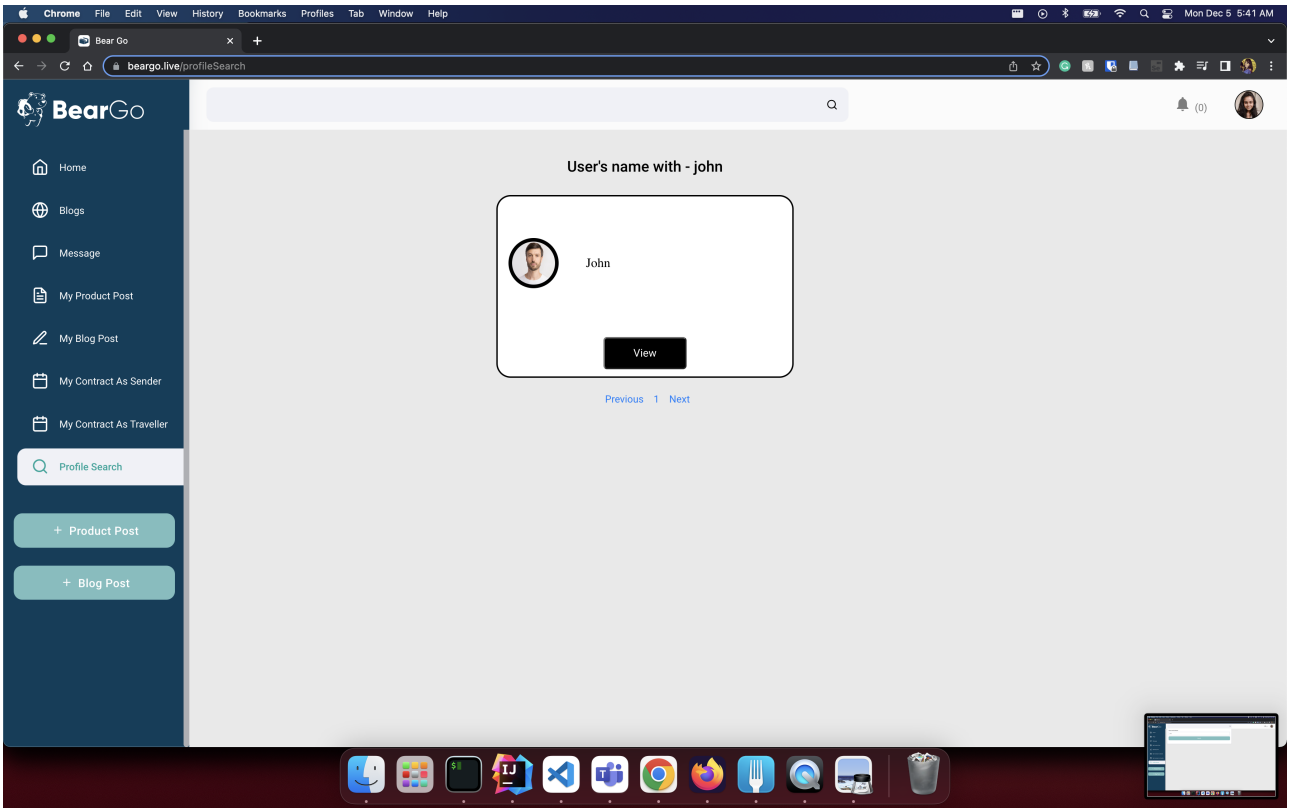
6.17 Traveler contract Page



6.18 User Search Page



6.19 List of User Found Page



7 Implementation

7.1 Backend

Our backend server is developed using Spring Boot. Core maven dependencies:

1. Spring Boot Web Starter
2. Spring Boot Data JPA Starter
3. Spring Boot Security Starter
4. Spring Security JWT Library
5. Lombok
6. JUnit Jupiter Engine
7. Spring Boot WebSocket Starter
8. PostgreSQL
9. H2 Database Engine
10. SendGrid Java
11. SpringFox Swagger UI
12. modelMapper
13. Spring Boot Kafka
14. Spring social twitter

7.2 Frontend

We used React.js to develop our frontend client. We chose React.js over other web development framework because its DOM is declarative and this Javascript library is efficient to use. We made connection between frontend and backend through REST APIs.
To run the project we used the following dependencies -

1. **react-router-dom** - It is used to route from one page to another in the application.
2. **axios** - It is a HTTP client library used to send asynchronous HTTP requests to REST APIs.
3. **react-star-ratings** - It is used to show star rating in the application.
4. **react-share** = It is used to show social media buttons and has functionality like share something in social media.
5. **react-stomp-client** - It is used to establish connection and subscribe in multiple topics.
6. **sockjs-client** - It is a Javascript library to open the Websocket.
7. **react-paginate** - It is used to handle pagination in the application.

7.3 Backend Authentication and Authorization

For our backend server's authentication and authorisation policies, we used Spring Security. When a user successfully logs in, the backend generates a JWT token and utilizes that token to authenticate subsequent rest requests. To grant access to specific controller methods, we implemented role-based access control. We did this by annotating controller methods with @hasRole.

7.4 Frontend Authentication and Authorization

A new user, when first come to our application, has access to the login page. There they will find the option to register or login into the system. While registering, to verify their email address, a token will be sent to the user's email address. Upon verifying the email address, a user can register to the system. Once registered and authentication, the user will be able to login to the system and will be able to see Productposts and blogs.

7.5 Email Verification

We have supported email verification system while registration to support one step verification for our users. The user name must be a valid email address for registering into our system. A confirmation code will be sent to this email email address and user has to enable the account by verifying through email. We are using SendGrid for this purpose.

7.6 REST API

Our REST endpoints were implemented using Spring Boot's Rest Controller. According to the requirements of our front end, we have created roughly many REST APIs in our backend. Some of the mentionable ones are resolveProductPost, createProductPost etc.

7.7 Password Reset

The forgot password option allows the user to reset his account's password. He must enter his registered email. He will receive a confirmation number via email, which he can use to create a new password. To produce and email confirmation codes, we used the same technique as during registration.

7.8 Websocket Implementation

We have implemented WebSockets in three use cases.

- peer-to-peer chat
- Sending Notification
- Tweet feed

In the backend, we send message through the web socket to a particular topic. The topic is created in the format **/topic/newmsg+[userid]**. Hence an unique topic created for each user. In the front-end, we implemented stompclient over sockjs. The stompclient is subscribed to the topic. Whenever a new message is arrived, it updates the message list with the latest message and also in the chatbox window, the message appears.

For sending notification, we also created another topic in a similar fashion **/topic/newNotification+[userid]**. Whenever send notification service is called in the backend, the notification is saved and also sent through the websocket. The front-end is subscribed to the topic. Whenever a new notification is arrived, it appends the notification the notification list at the start and increases the counter.

For Twitter, we implemented Twitter version1 stream api. Whenever a tweet with beargo is received in the streamed api, it sends thorug the web socket with fixed topic **/topic/tweet**. The front-end is subscribed to the topic. Whenever a new twwet is arrived, it appends the tweet int the tweet list at the start.

7.9 Kafka Implementation

Kafka has been used in three use cases as well.

- peer-to-peer chat
- Sending Notification
- Tweet feed

For messaging, whenever a message is sent by the user, the message is saved in the database and also sent to the kafka. We have kafka producer configured to send message object to the kafka. We also configured kafka listener for message object with the corresponding topic, so that it can listen and can deserialize the object. Once the message is received, the message is sent through the web-socket.

For notification, internal service calls the save notification service when notification is required to be sent. In the service, the notification is saved in the database, and also sent to the kafka. We have kafka producer configured to send notification object to the kafka. We also configured kafka listener for notification object with the corresponding topic, so that it can listen and can deserialize the object. Once the notification is received, the notification is sent through the web-socket.

In similar fashion, we implemented the kafka for twitter. Details are given in the following section

7.10 Twitter Implementation

For twitter, we created a separate application which is independent of others. We used twitter4j library for our application. We implement twitter version filtered stream api. The keyword we defined is **beargo**. Since it is a separate application we had to define the package name in the listener as a trusted package, so that no security conflict arises. Also, we added user-type-info-headers false. So, if anyone tweets with the keyword beargo, the data is available in stream api. Then we send that data to the kafka.

In our main application, we configured our kafka listener with the topic and for the object twitter to be deserialized. The same class is declared in both the application. Once the listener receives the data, it saves the data in the database and sends over the web socket. Front-end stompclient subscribed to the topic, gets the data and shows it, using the react-twitter widget. This widget only requires the twitter id, and then it displays the data.

7.11 Development and Production Profiles

For the development and production environments, we handled different Spring Boot profiles. For development, we used the H2 database, and for production, MySQL.

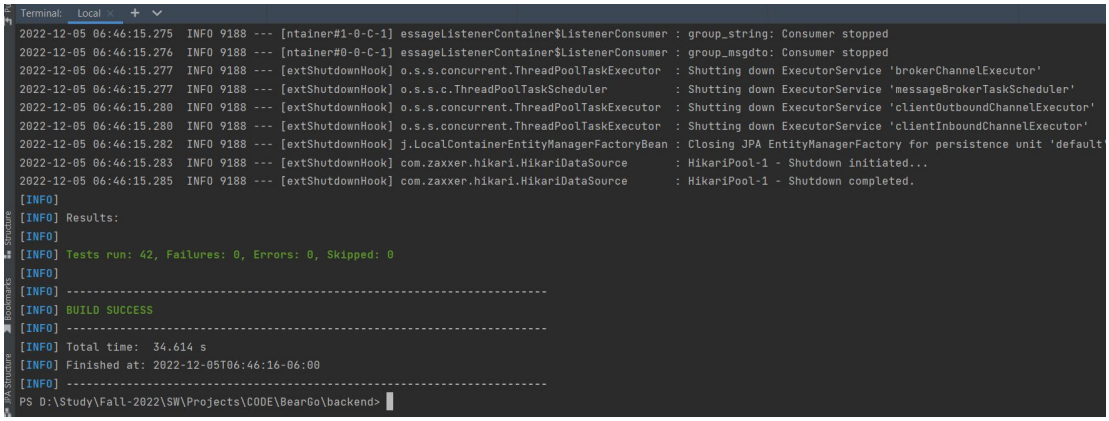
7.12 Logging

We implement ELK stack for the logging purpose. All the api request log and response time are sent to logstash. This will help to monitor the services' health. According to the health,
Kibana visualization tool helps us to debug any issue. All the logs can be visualized and can be searched by the package name.

8 Testing and Debugging

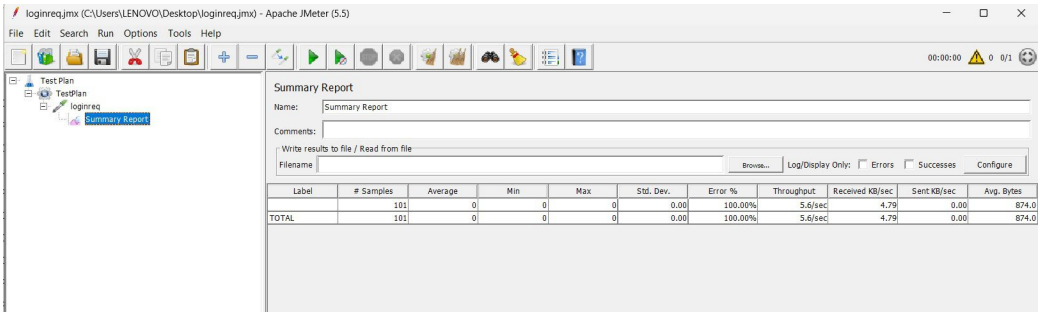
8.1 Unit Test

For our project, we used Junit testing. We tested our controllers and service through JUnit testing. We have 42 test cases altogether.



8.2 Load Testing

We also used JMeter for the load testing. Following is the output provided by the JMeter.



9 Deployment

Our application is deployed in DigitalOcean Droplet using Docker Compose. The Docker Compose files wraps all containers together and manages networking between the containers.

9.1 Server Configuration

- OS: Ubuntu 22.10 x64
- Memory: 8 GB
- CPU: 2 Dedicated General Purpose vCPU
- Disk: 10 GB

9.2 Backend

First we packaged our SpringBoot application into a JAR file. Then we used 'openjdk:8' base image to prepare the Docker image of our backend. We passed environment variables to replace 'application.properties' dynamically. For example, by default we configured H2 database, but in production we replaced it to MySQL using 'SPRING_DATASOURCE_URL' environment variable.

9.3 Frontend

First we created an optimized production build for our React frontend application using 'npm build'. We used a 'node:12.4.0-alpine' image for the production build and 'nginx:1.21.6-alpine' image to serve the contents. The Nginx server also handles routing between frontend static files and backend APIs, this allowed us to use a single domain for backend and frontend.

9.4 Database

We used MySQL as our production database. Our database is also runs in Docker. In order to prevent data loss when Docker container restarts, we used a persistent volume. Also, to ensure security, our database is not exposed outside Docker. Only the backend can access the database using internal Docker network.

9.5 Kafka producer for Twitter

We used a separate module for asynchronously fetch Tweets and push it to Kafka. We used 'openjdk:17' base image to Dockerize the module.

9.6 Kafka Server

Our Kafka server runs as a standalone Docker container along with the required Zookepper container. Our backend consumer and Twitter producer communicates with the Kafka server using internal Docker network.

9.7 Logging

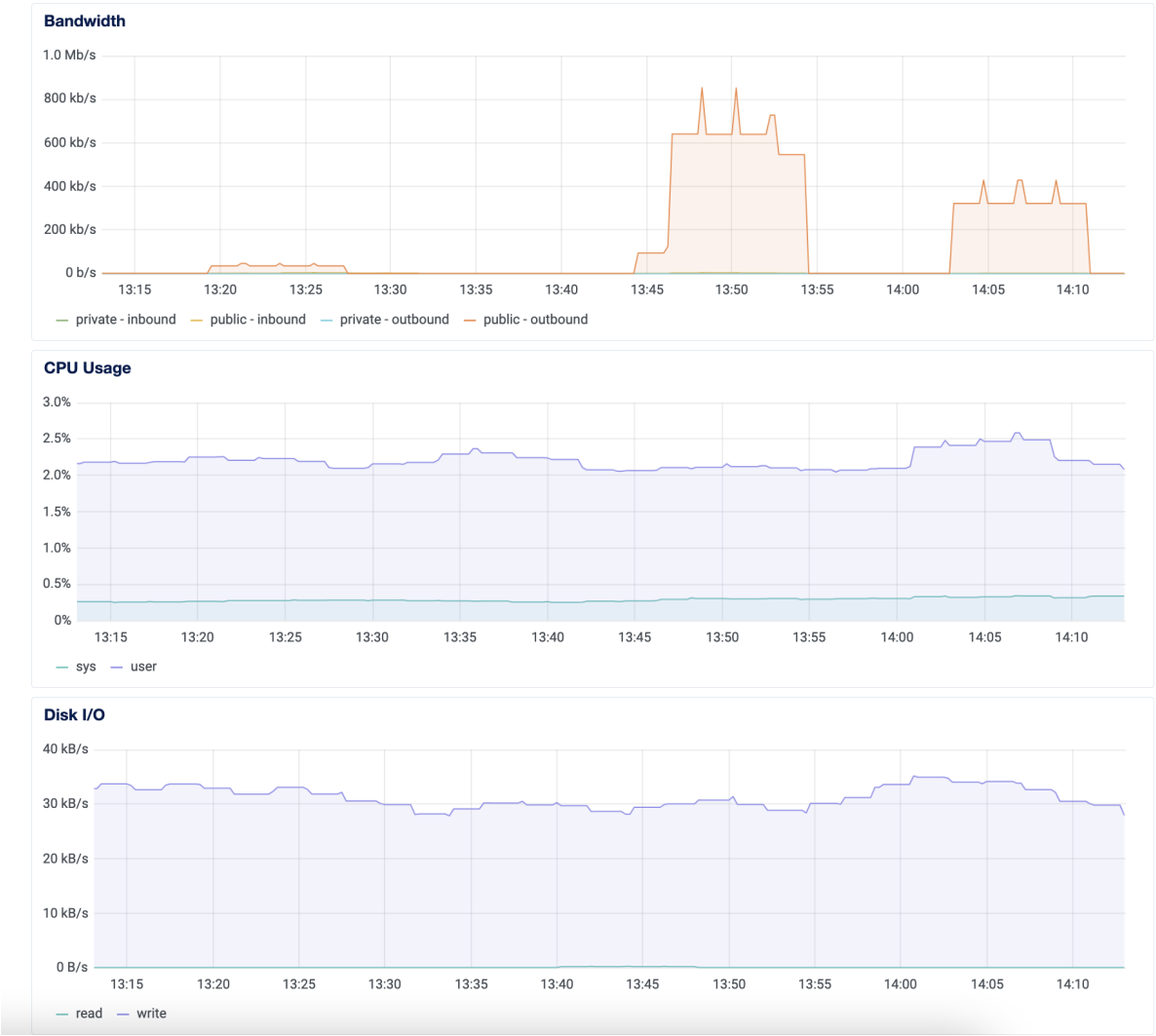
We integrated a Dockerized version of 'Kibana' in our application for log analysis. We used 'logstash' container to collect the logs and 'elasticsearch' container store the logs.

9.8 SSL integration

We used Let's Encrypt to get free SSL certicates for our 'beargo.live' domain and used Nginx (in frontend Docker container) to handle the https connection.

9.9 Monitoring

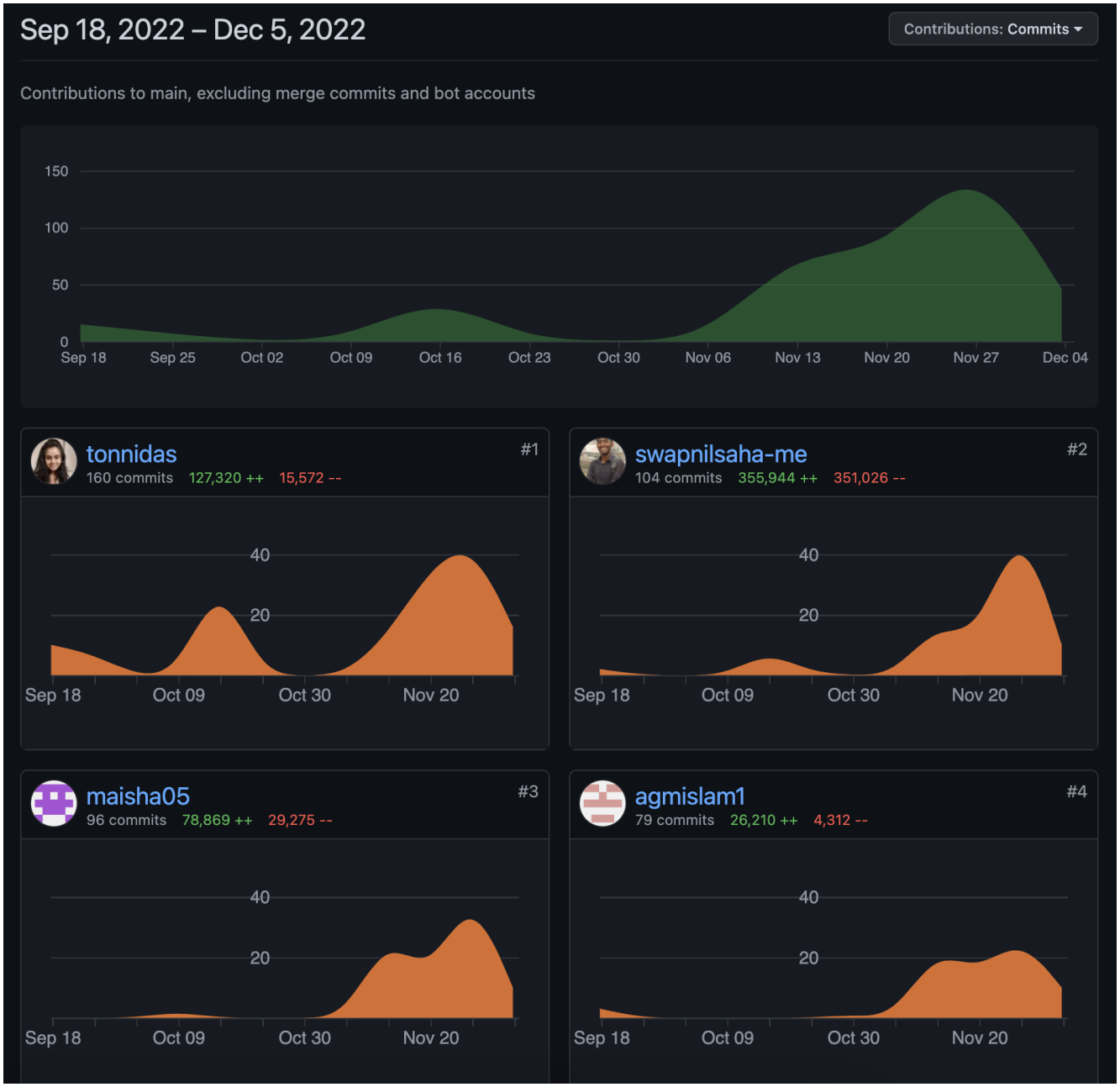
- We utilized DigitalOcean's monitoring tools for server health monitoring. Initially, we choose 4GB memory, then we found that some notification messages were not properly propagated through Kafka. Then in the memory monitoring we found that 98% memory is used. Then we switched to 8GB memory which resolved the issue.
- We also used free online monitoring tool Uptime Robot to monitor our website availability.



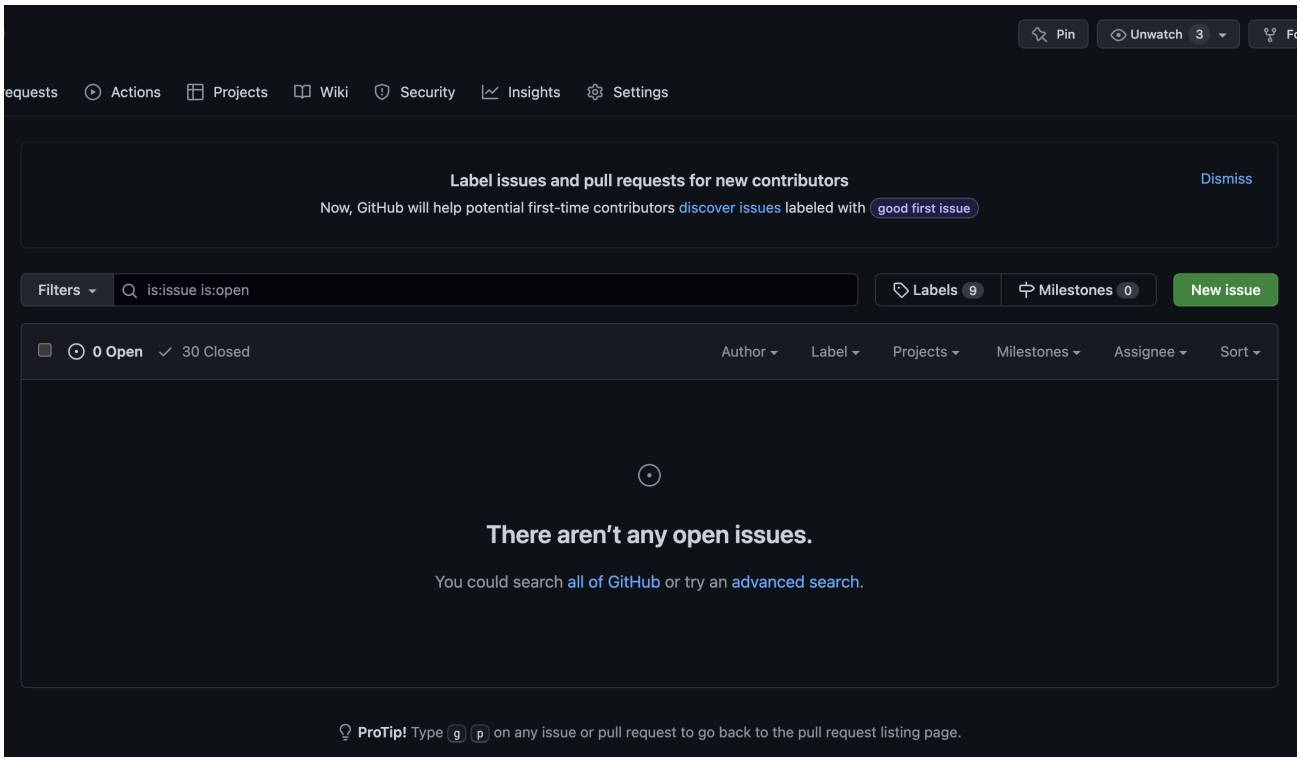
10 Git Summary

Our Github repository link: https://github.com/tonnidas/BearGo

Together we pushed about 440 commits in our Github repository.



We have a total number of issues are 30. We closed about all issues.

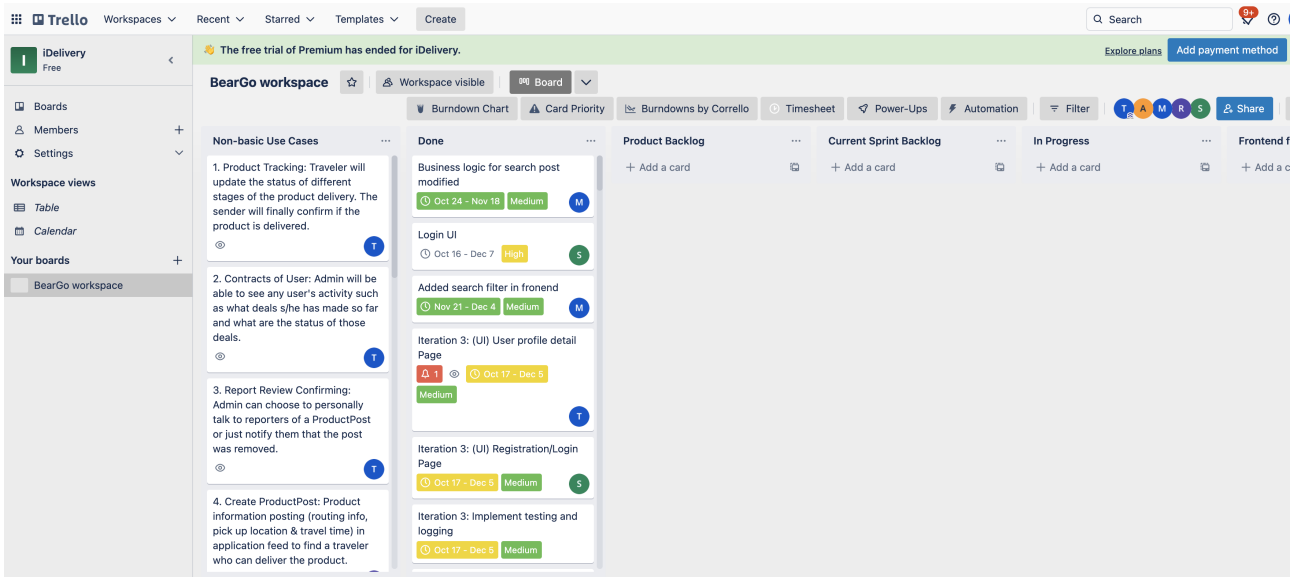


11 Cost Estimation

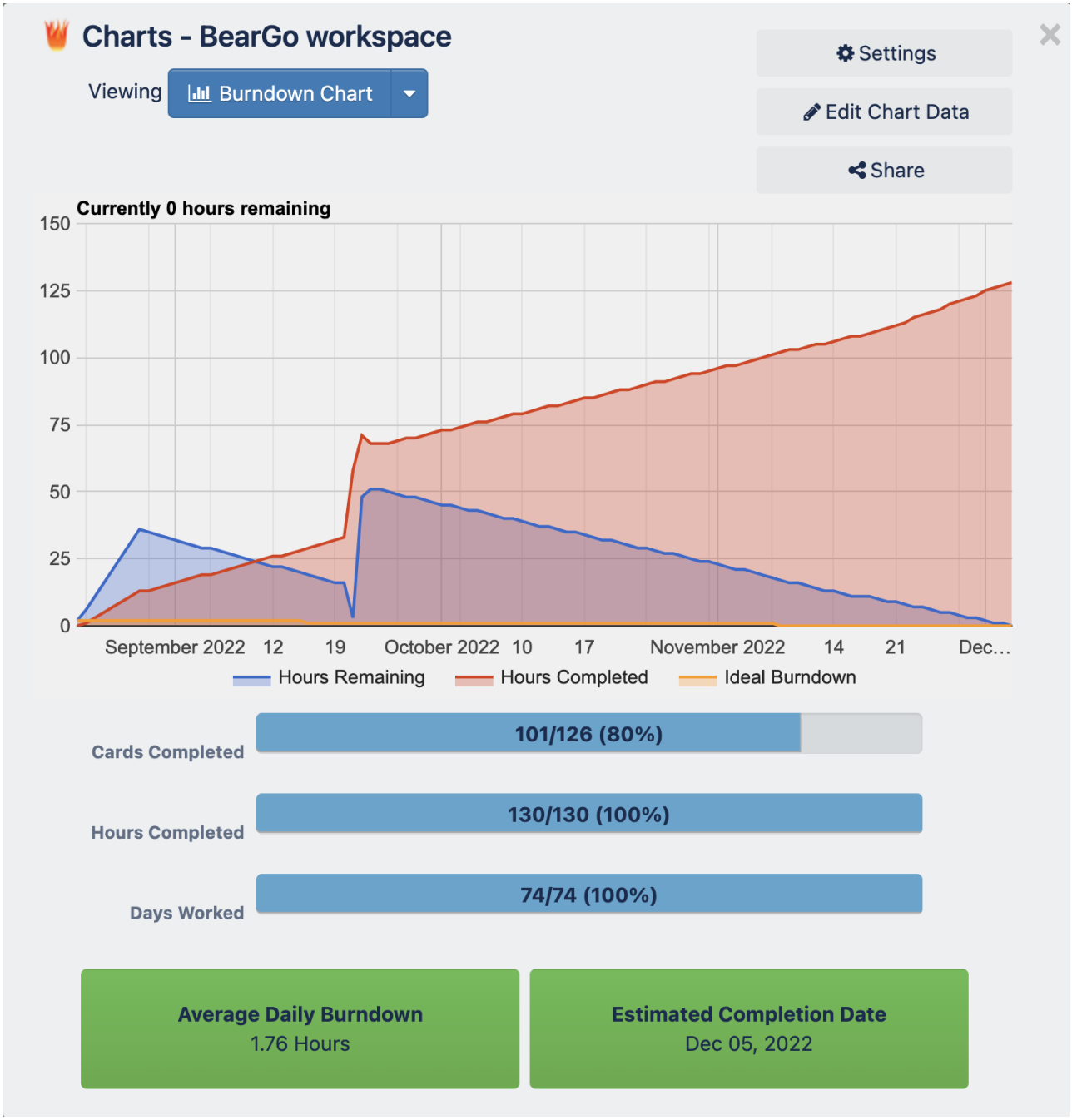
Each of us spent about 185 hours on this project. Followings are the breakdown for each iteration. Total work hour is 742 hours.
We assume that our average hourly cost is $40. Therefore, the total cost is **$29,680**.

12 Trello Updates

12.1 Trello Overview Overall



12.2 Trello Burndown Chart



12.3 Individual Trello Task View

12.3.1 AGM Islam

Task	Hours
Daily Scrum	17.5
Iteration 1: Complete task assignment in Trello for current sprint to team members (5)	1
Iteration 1: Create GitHub repository (1)	2
Iteration 1: Data Model(3)	3.92
Iteration 1: Deciding frameworks (backend and frontend) and create a skeleton project (6)	2
Iteration 1: Deciding use cases with business constraints (3)	1
Iteration 1: Designing Activity Diagram (3)	2.66
Iteration 1: Designing Fully addressed use cases (10)	2.5
Iteration 1: Designing system operations with operation contracts (3)	2
Iteration 1: Designing system sequence diagram (SSD) (4)	4
Iteration 1: Drawing use case diagram (3)	7
Iteration 1: Identify entities and create class diagram / Domain model (10)	3.25
Iteration 1: Identifying the timeline for each iteration (3)	0.5
Iteration 1: Prepare Gantt Chart (5)	1
Iteration 1: Requirement refining (6)	1
Iteration 1: Wireframes & html (3)	8
Iteration 2: Demo with CRUD REST API (12)	10
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
Iteration 3: Implement testing and logging	3
Kafka Config	2
Kafka: Listener, Producer	4
Message Controller	3.5
Message Service	4.5
Model: Message, Notification	2.5
Notification	5
Notification Controller	4
Notification Service	4
Peer-peer chat	9
Security Config	5
Sprint Retrospective	14
Sprint Review	5.65
Twitter Implementation	14
WebSocket Config	4
Total	**185.98**

12.3.2 Maisha Binte Rashid

Task	Hours
Added My contracts as Sender in the frontend.	2.25
Added My contracts as traveller in the frontend.	2.5
Added search filter added in backend.	2.17
Added search filter in fronend	2
API connection created from frontend to backend	3
API created for getting contracts.	1.75
API for Ban user created	1.75
Backend developed for My contracts as Traveller.	0.5
Business logic for search post modified	2.25
Business logic modified for ban user	2.5
Cost showing and interested people updating in Contract frontpage	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	17.5
Delivery status changed and cost added in Contract pages.	2
Frontend home page added	4.5
Homepage design modified	4.5
Iteration 1: Complete task assignment in Trello for current sprint to team members (5)	1
Iteration 1: Drawing use case diagram (3)	2
Iteration 1: Identify entities and create class diagram / Domain model (10)	1.75
Iteration 1: Identifying the timeline for each iteration (3)	1
Iteration 1: Prepare Gantt Chart (5)	1
Iteration 1: Requirement refining (6)	1
Iteration 1: Wireframes & html (3)	3.5
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	8.5
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
modified Search criterions in backend.	3.25
Notification bar design modified	2.5
Notification design added	3
Productpost updatd filter added in backend.	2.5
Productpost updated added in frontend.	1.75
Register page added	2
Register page modified	2.5
Sprint Retrospective	14
Sprint Review	6.15
Total	**175.23**

12.3.3 Swapnil Saha

Task	Hours
8. Ban User: User can report against another user and admin is responsible to ban a user from	2
Add Ban User	4
Add Invoice	4
Add Review Rating	7
Admin Profile Page	4.5
BlogPost Creation	9.25
Daily Scrum	17.75
Iteration 1: Complete task assignment in Trello for current sprint to team members (5)	6
Iteration 1: Create GitHub repository (1)	2
Iteration 1: Data Model(3)	3.7
Iteration 1: Deciding frameworks (backend and frontend) and create a skeleton project (6)	1.83
Iteration 1: Deciding use cases with business constraints (3)	1.08
Iteration 1: Designing Activity Diagram (3)	2.85
Iteration 1: Designing Fully addressed use cases (10)	2.5
Iteration 1: Designing system operations with operation contracts (3)	2
Iteration 1: Designing system sequence diagram (SSD) (4)	8
Iteration 1: Drawing use case diagram (3)	2.92
Iteration 1: Identify entities and create class diagram / Domain model (10)	3.67
Iteration 1: Identifying the timeline for each iteration (3)	0.42
Iteration 1: Prepare Gantt Chart (5)	1.17
Iteration 1: Requirement refining (6)	0.67
Iteration 1: Wireframes & html (3)	8
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)	8
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
Search User	4
Sprint Retrospective	14
Sprint Review	6.65
User Profile Page	8.5
Total	**190.23**

12.3.4 Tonni Das Jui

Task	Hours
Authentication system	4.2
Blocking a productpost from admin panel	2
BlogPost page	1
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
Collapsible comment section	2
Creating BlogPost page	2.5
Creating productPost page	5
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	18.25
Email verification for registration and password reset	1
Image data fetching in home page	1.5
Iteration 1: Complete task assignment in Trello for current sprint to team members (5)	0.67
Iteration 1: Create GitHub repository (1)	4
Iteration 1: Data Model(3)	5.87
Iteration 1: Deciding frameworks (backend and frontend) and create a skeleton project (6)	2
Iteration 1: Deciding use cases with business constraints (3)	2.08
Iteration 1: Wireframes & html (3)	6
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	13.2
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
Modifying a productPost(for admin option)	1
ProductPost editing page	2
ProductPostPage data fetching from backend	2
Showing interest on a ProductPost	2
Sprint Retrospective	14
Sprint Review	6.15
Writing Unit test for services	2.5
Total	**192.68**

13 Demo

Video Link: https://www.youtube.com/watch?v=RovVsEsy4Fo

14 JavaDoc and Swagger API Document

1. Our Swagger API document is available here: Swagger API Documentation
2. Our javaDoc document is available here: Generated Java Docs