Advanced Software Engineering
FINAL REPORT

Team number:	0501

Team member 1	
Name:	Hynek Zemanec
Student ID:	12010957
E-mail address:	a12010957@unet.univie.ac.at

Team member 2	
Name:	Christian Lascsak
Student ID:	01363742
E-mail address:	a01363742@unet.univie.ac.at

Team member 3	
Name:	Preisinger Johannes
Student ID:	01106089
E-mail address:	a01106089@unet.univie.ac.at

Team member 4	
Name:	Kapeller Angelika
Student ID:	01268478
E-mail address:	a01268478@unet.univie.ac.at

Team member 5	
Name:	Puri Himal
Student ID:	11738090
E-mail address:	a11738090@unet.univie.ac.at

1. Final Design

1.1. Design Approach and Overview

1.1.1. Assumptions

Project goal

Within the last few years, craft beer has become very popular in Austria. However, special brands of craft beer are often only limited to a small range of bars or mainly sold in regional supermarkets. Therefore, a lot of brands are still widely unknown to most beer lovers. For this reason, the goal of this project is to introduce a platform for selling and buying locally brewed craft beer to satisfy the ever-growing demand for new tastes among craft beer fans.

Target Group - Customers

The webshop's target customers are people between the age of approx. 20 and 35 who love craft beer and have experience with ordering online. Customers of the webshop not only enjoy drinking beer but want to experience new flavours and get to know new craft beer brands. The webshop enables them to effortlessly order their favourite craft beer brands and get them delivered directly to their home. Furthermore, the webshop provides the opportunity to discover new tastes, brands and breweries by online exploring the product list.

Target Group - Vendors

Vendors are small or medium craft beer breweries located in Vienna or Lower Austria. The webshop enables them to sell their products directly to the customers and therefore, to establish a close relationship to their customer base. Consequently, they can adapt their products to the needs of their customers and can profit from getting direct feedback on their latest flavours and seasonal brands.

Location

The webshop is currently designed for selling products in Vienna and Lower Austria. This enables stable shipping times and a close relationship between customers and local breweries.

Limitations and future goals

- As a future goal, the webshop could expand to selling products in all federal states of Austria or even neighbouring countries.

- The webshop currently provides textual information only in English. As a future goal, the website can be translated to German and other languages as demand increases.
- The only available payment method is payment on delivery. With a growing scale of the webshop, other payment methods will be offered.
- The website will be tested with selected customers and vendors for two months. During this time, vendors and buyers can not create an account on the website on their own. By writing an email to the webshop company, they will receive their user credentials to log in to the webshop. After the testing period is successfully completed, the website will be open for everyone and the registration process will be added.

Stakeholders

- Webshop company
- Breweries
- Customers
- Transport suppliers
- Competitors
- Client competitors
- Developer team
- Operations
- DevOps
- Test team
- Marketing
- Legal department
- Customer Service

1.1.2. Design Decisions

Changes in our service landscape

At first, we created the system without a notification Service in mind. In this case, each microservice would have had their own Mail Server, that is used for sending notifications to users.

During the design phase however, we noticed that the notification logic is actually its own bounded context. We therefore separated it from the other services and created a single microservice for it. This better separates concerns between services that do business logic, and a service that notifies the users. Further, we have also decided to add individual databases to each microservice.

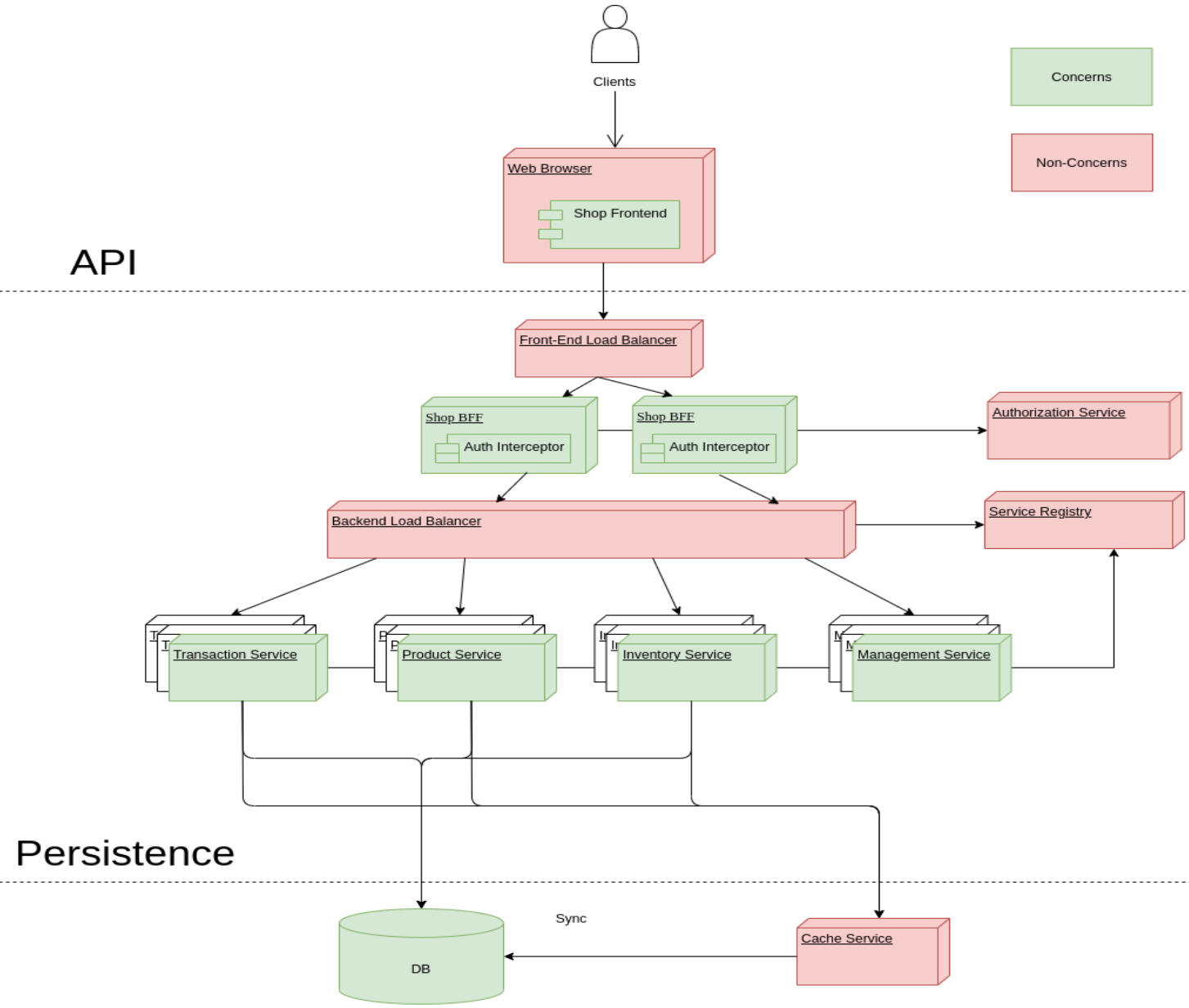


FIG: Old version, with no Notification Service

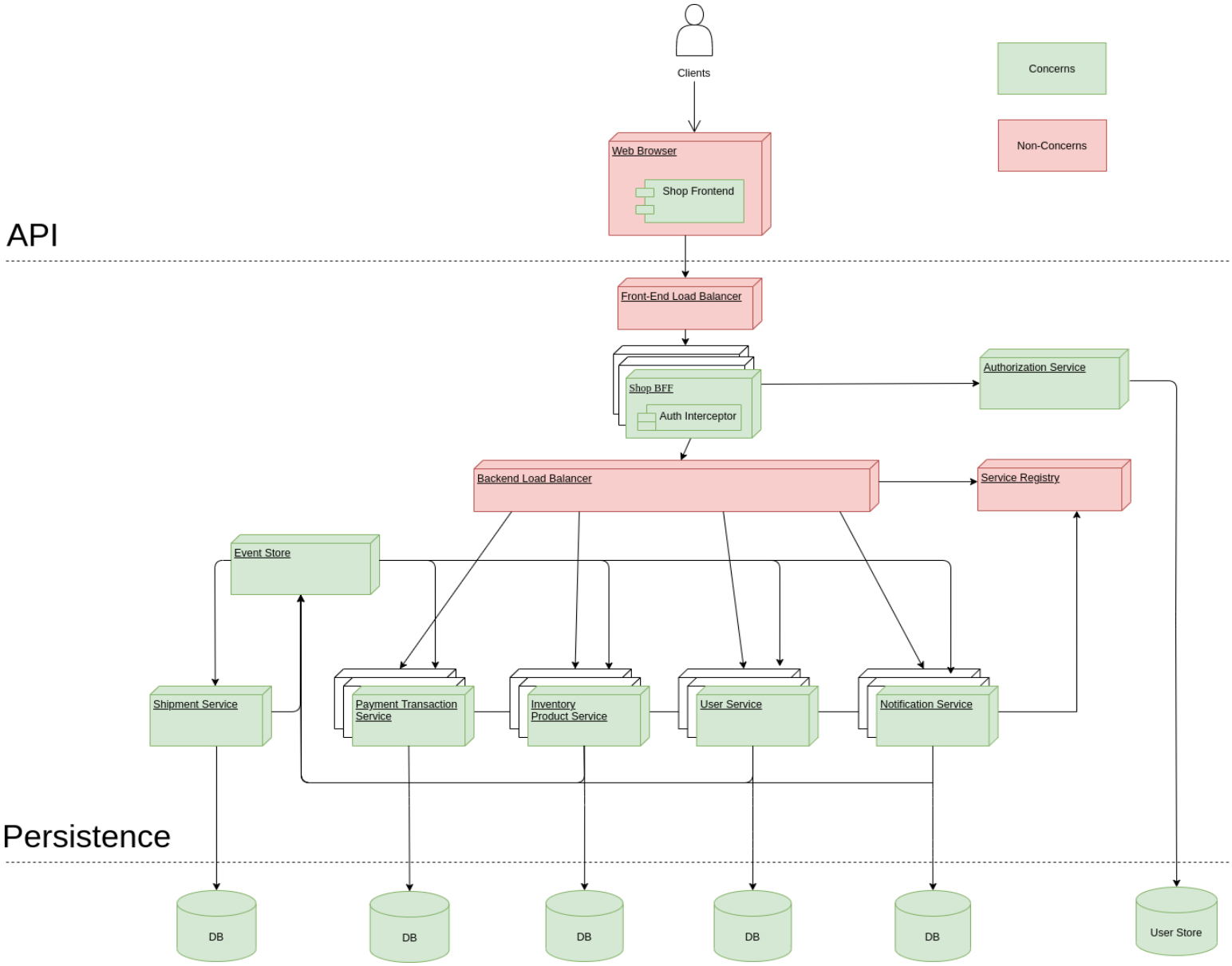


FIG: Design Report version with Notification Service

FIG: Final version

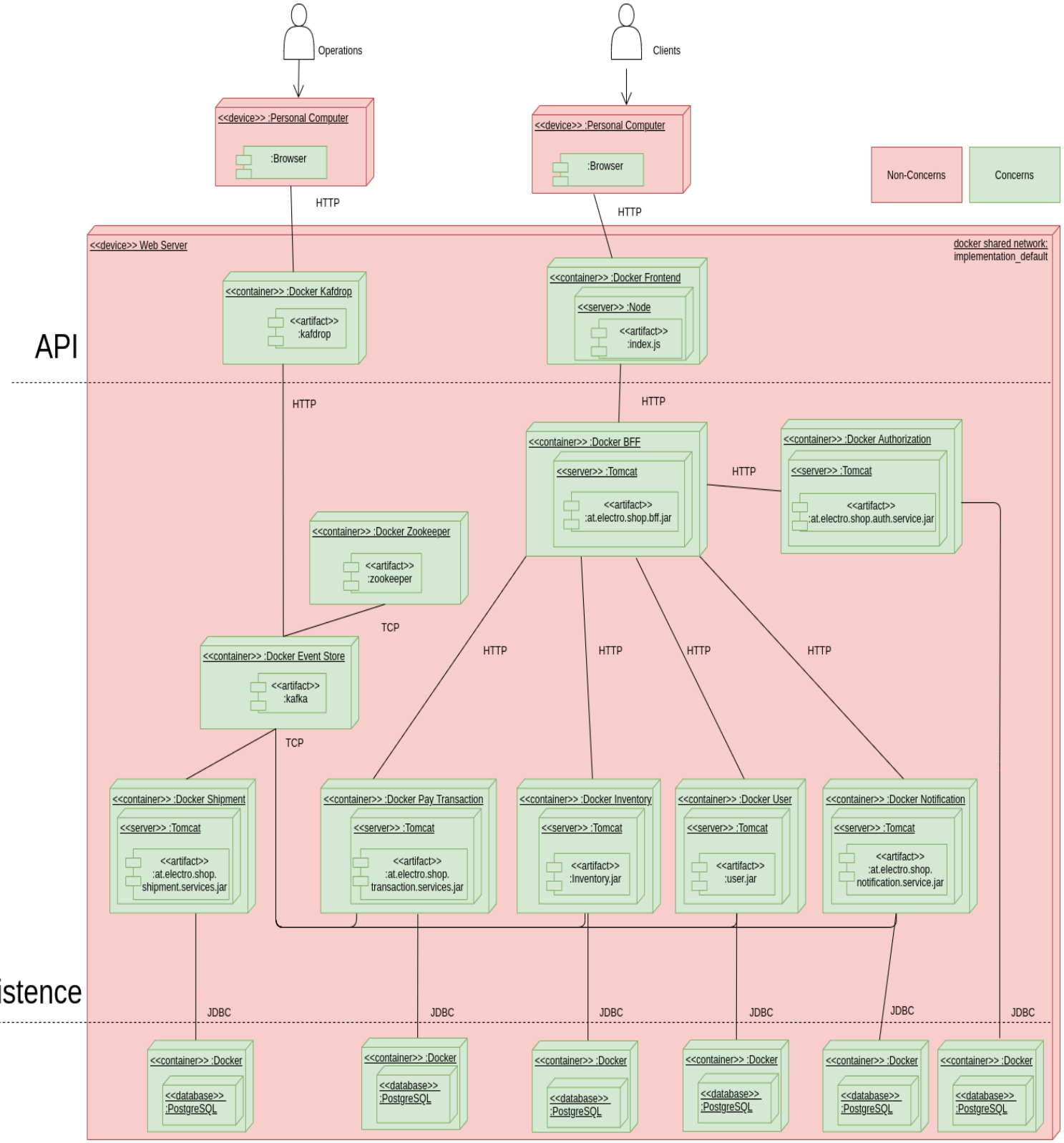


FIG: FINAL VERSION

In the most recent version we have also decided to document the landscape of our event store in more detail, and which artifacts we actually deploy to the different systems. We also wanted to highlight that the complete solutions actually share one network, and how the applications communicate over this network.

User Microservice vs No User Microservice

Due to the fact that we thought we could store customer related data in the Authorization Datastore, we thought we could also store all customer related data in the Authorization Database. But because we decided it would be a better separation of concern to separate the Authorization Domain from the User Domain and therefore also detach the Account from the User, we actually created two different deployable services and therefore separated datastores. The Authorization service and the User service. We made this decision, because we found during the design period that the Authorization Domain should be ignorant of the kind of system it is used for. It should be reusable to future systems of different design and we will offer it as an injectable Interceptor to future projects of this kind.

Synchronous vs Asynchronous Communication for Querying Data

Firstly, we designed the querying for products in an asynchronous way, via the message broker. This means that a message would be published, requesting the products. The product service would consume this event and publish a new one with the queried products (see Process View).

However, this process is not good event driven design. Therefore we changed it to be a normal REST request.

BFF does not communicate with message Broker

During the design phase, the BFF (Backend for frontend) was communicating with the message broker directly. It would therefore publish certain messages that will be listened to by the other microservices. This means that messages would be published based on user interactions.

This approach has its drawbacks, since the frontend needs synchronous communication. The user needs to know if an action was successful or not (e.g. buying a product). We therefore decided to let the BFF communicate with the services directly. However, the services can only communicate event-driven with each other. This still ensures that our core domains or bounding contexts don’t have to know each other.

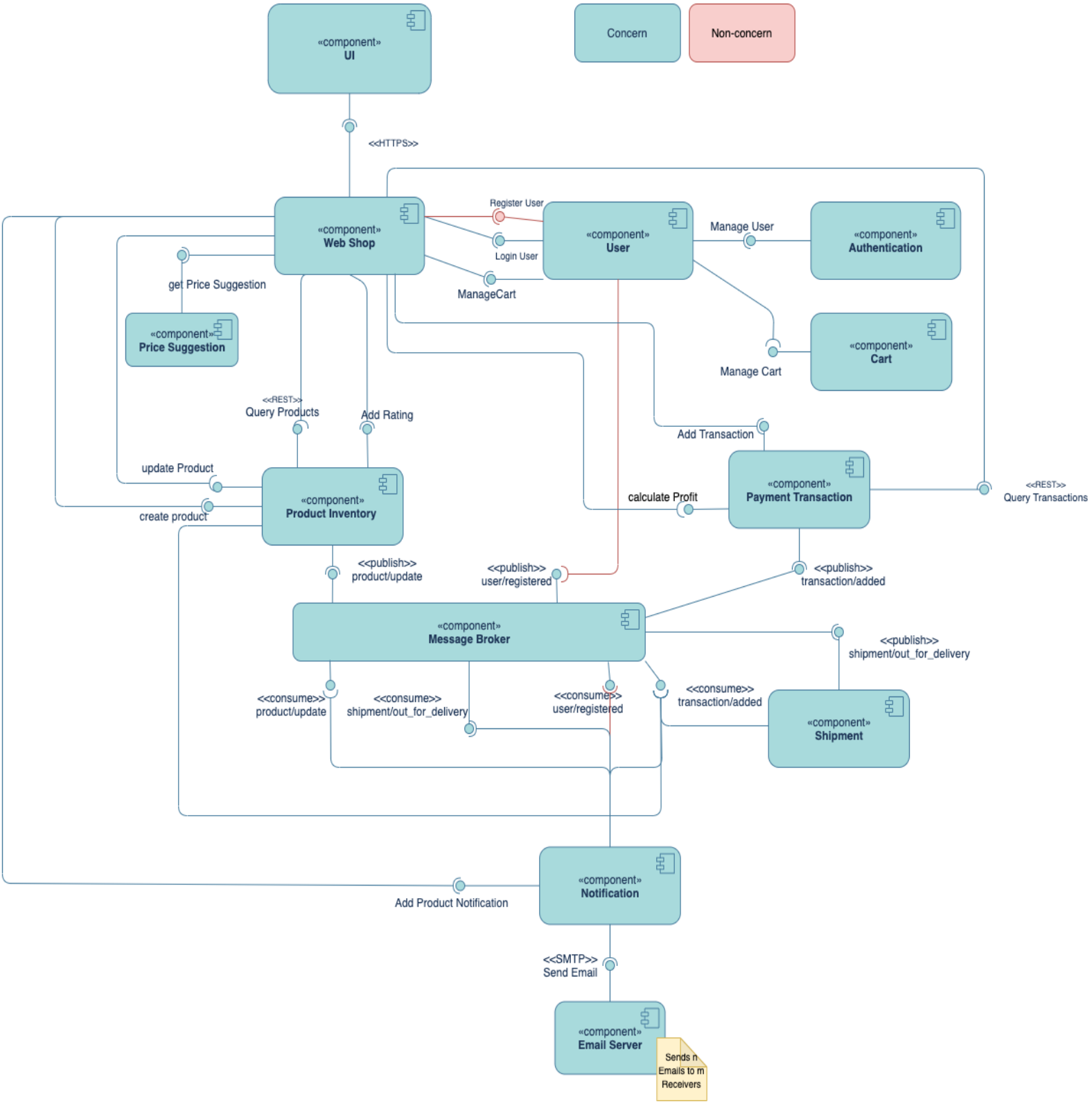


FIG: Old version, with BFF also publishing events

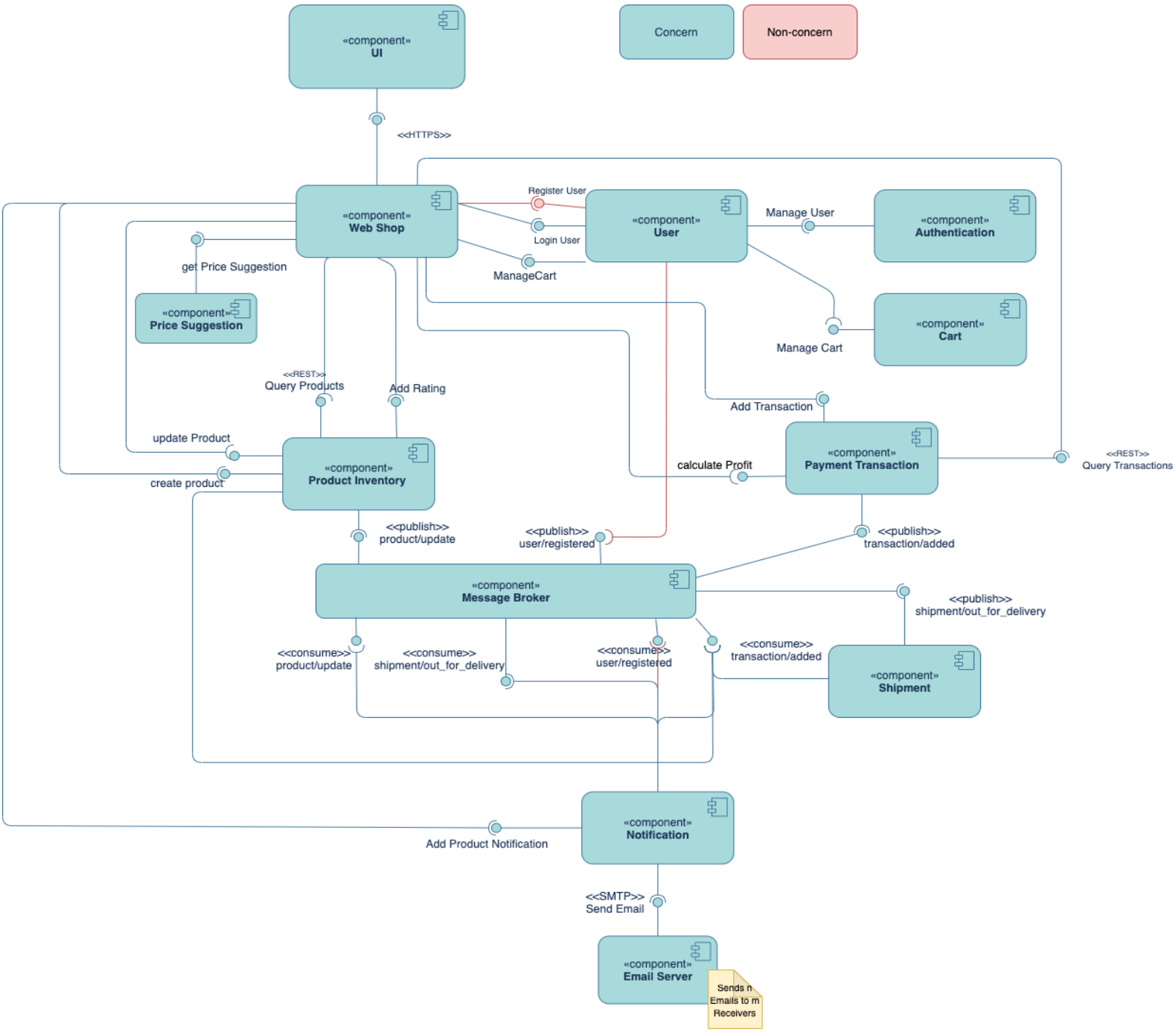


FIG: New version, with BFF only calling REST calls

Synchronous vs Asynchronous Communication

In the first iteration, we designed the system to communicate completely synchronously with REST calls. We noticed in our component diagram, that it is very complex and has no easily followable flow.

After discussions, we decided to use Kafka as a message broker for our system. Another framework we could have used was the java message service (JMS). We decided to go for Kafka, since it is the more modern and now widely used approach.

The changes can are reflected in the Component Diagram:

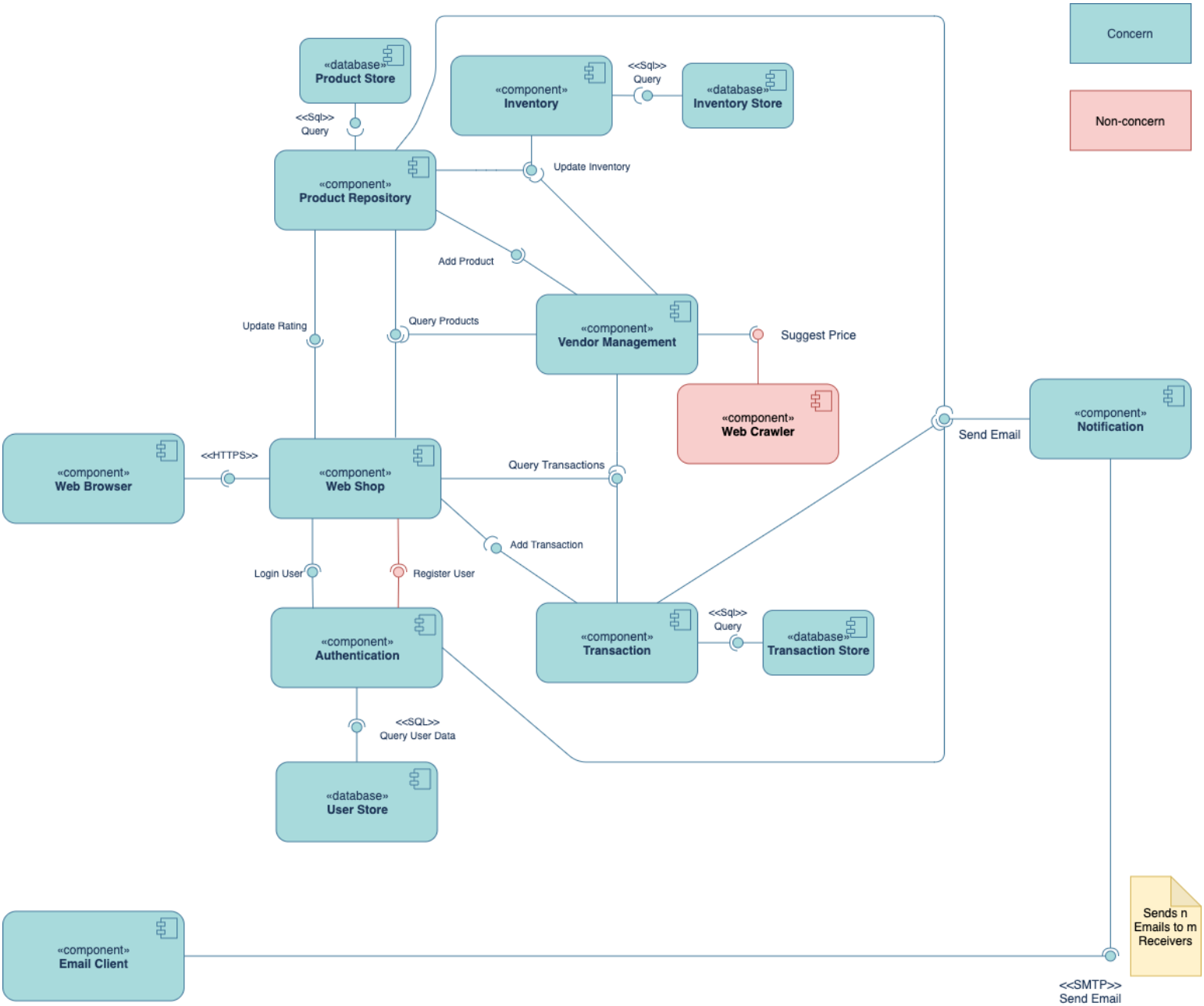


FIG: Synchronous Messaging

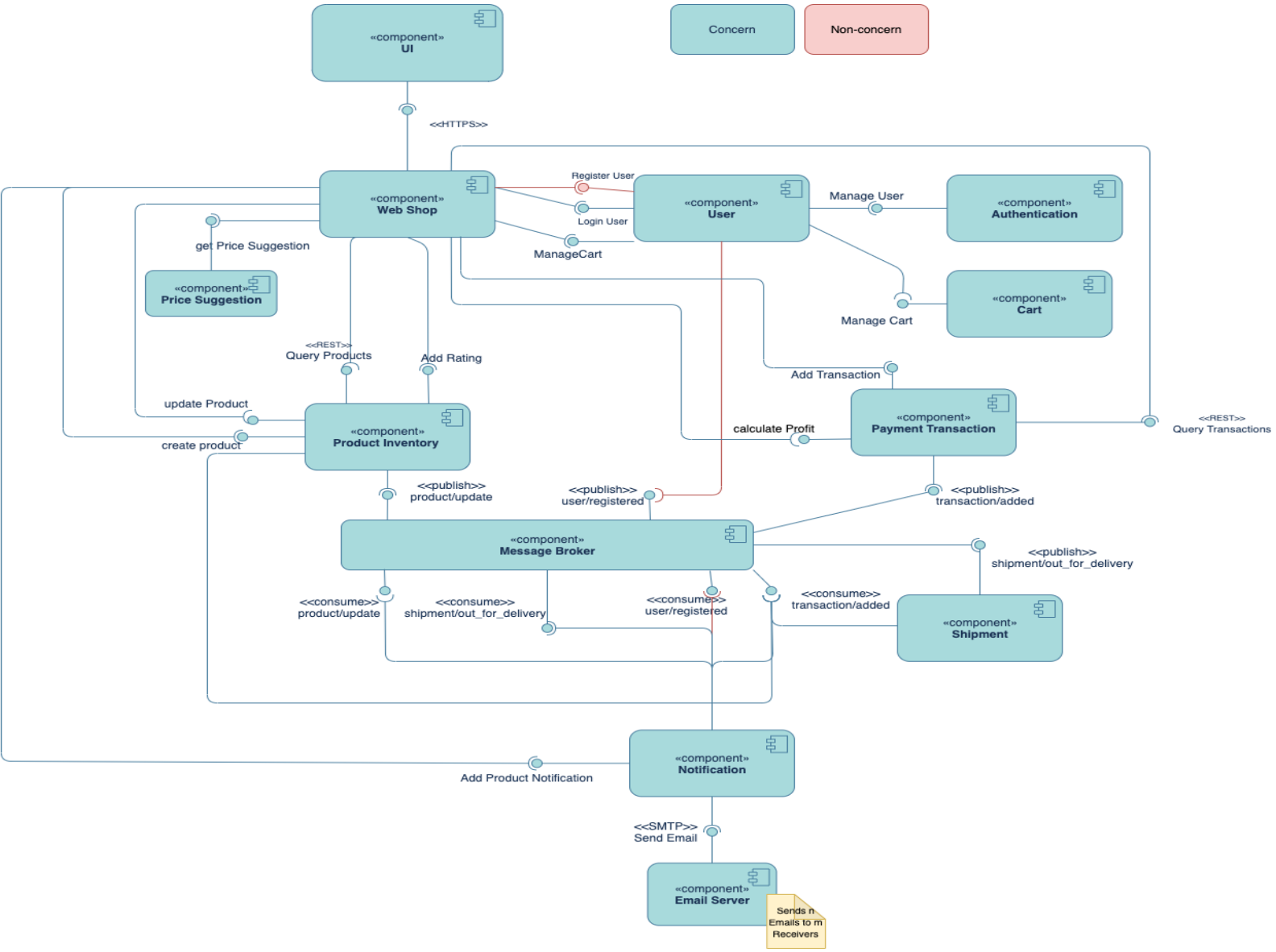
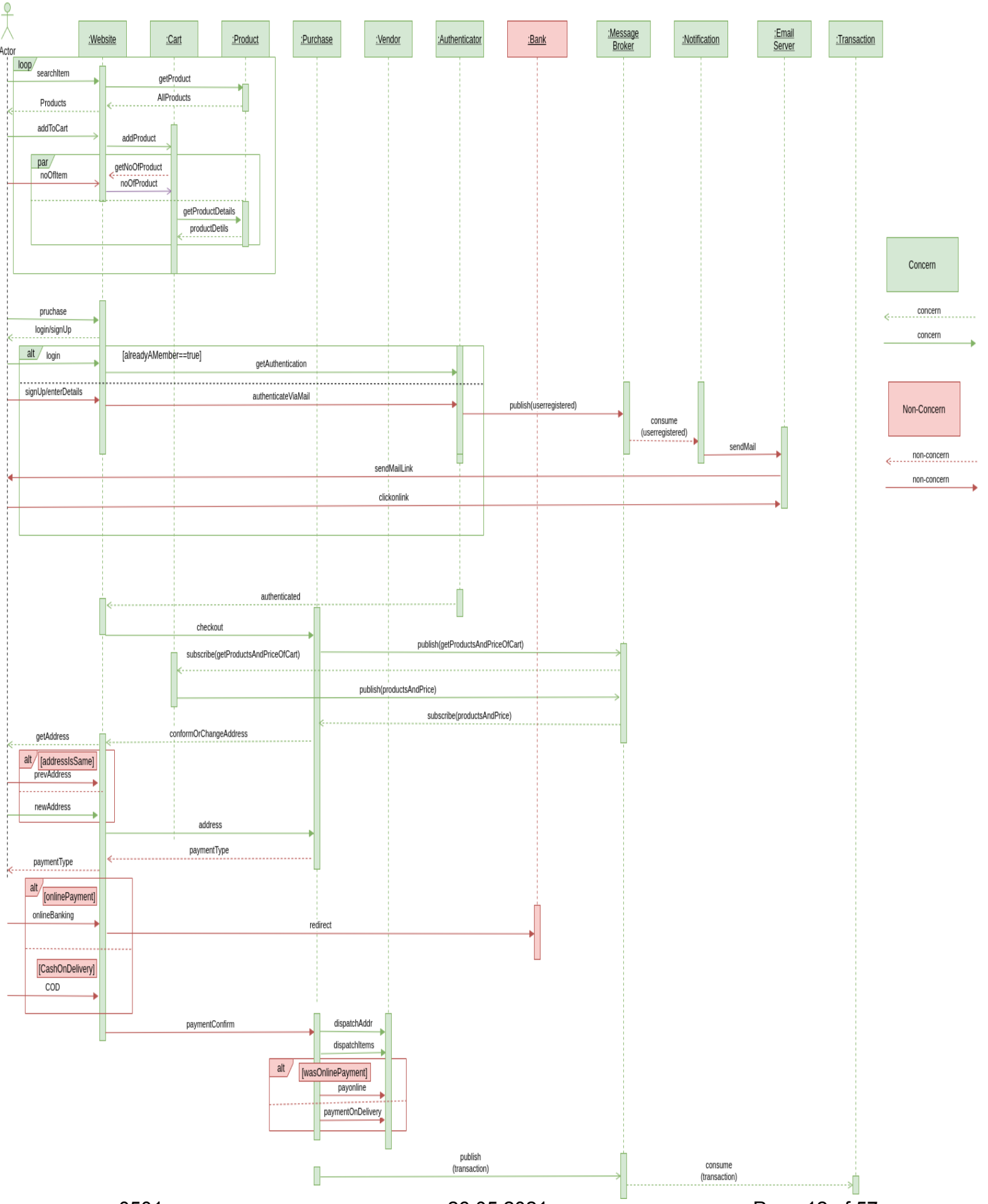


FIG: Asynchronous Messaging

Iteration of sequence diagram over the design phase.

The sequence diagram was built with the REST communication during the initial phase. Due to the need for event driven design, we have iterated it 4 times and come up with the final design. The last three are with event driven design with some adjustment due to the need to change for communication. The first diagram shows the rest api communication sequence diagram. The second one is with a kafka message broker that will have asynchronous communication with the other MS. The image below is the picture of the 4th iteration of the diagram in the Design phase. The new version is divided and placed in process view in section 3.3.

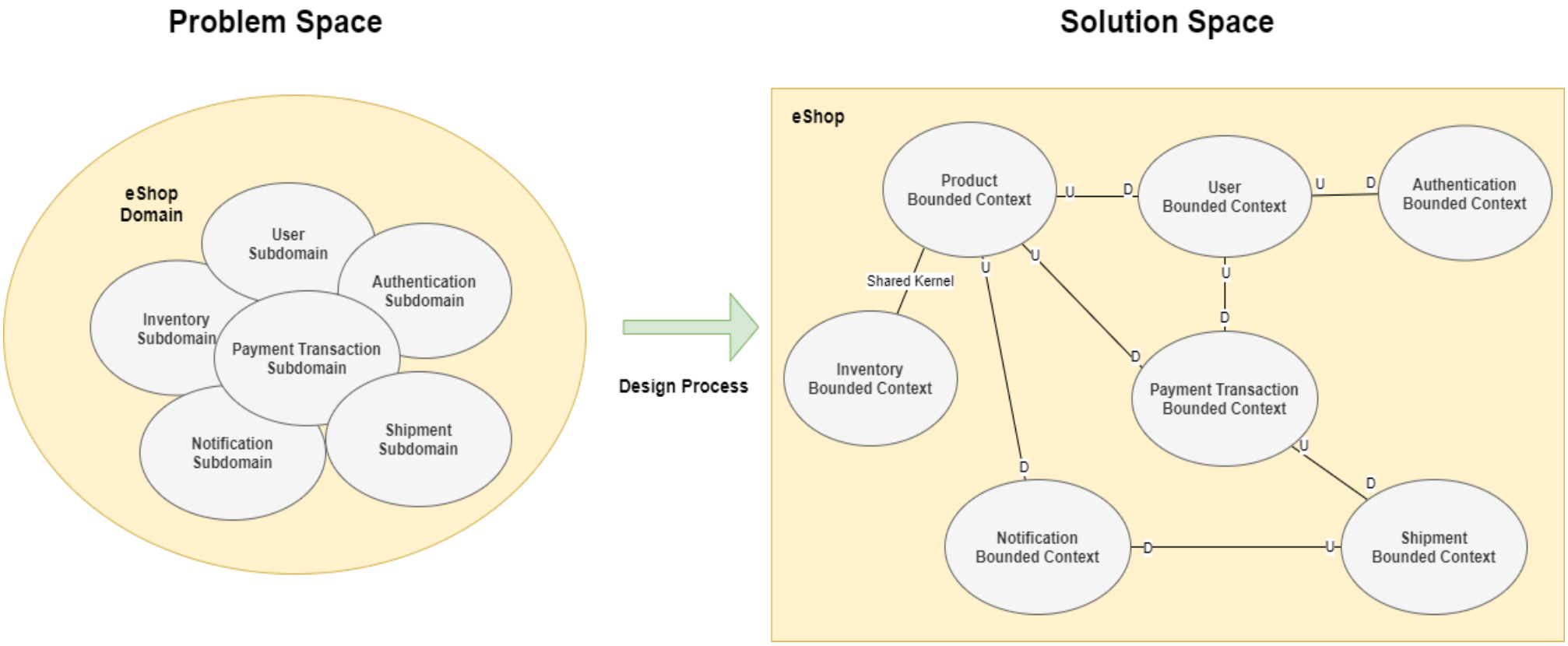

Actor
:Website
:Cart
:Product
:Purchase
:Vendor
:Authenticator
:Bank
:Message Broker
:Notification
:Email Server
:Transaction
loop
searchItem
getProduct
AllProducts
Products
addToCart
addProduct
par
getNoOfProduct
noOfItem
noOfProduct
getProductDetails
productDetils
Concern
concern
concern
pruchase
login/signUp
alt
login
[alreadyAMember==true]
getAuthentication
signUp/enterDetails
authenticateViaMail
publish(userregistered)
consume (userregistered)
sendMail
Non-Concern
non-concern
non-concern
sendMailLink
clickonlink
authenticated
checkout
publish(getProductsAndPriceOfCart)
subscribe(getProductsAndPriceOfCart)
publish(productsAndPrice)
subscribe(productsAndPrice)
getAddress
conformOrChangeAddress
alt
[addressIsSame]
prevAddress
newAddress
address
paymentType
paymentType
alt
[onlinePayment]
onlineBanking
redirect
[CashOnDelivery]
COD
paymentConfirm
dispatchAddr
dispatchItems
alt
[wasOnlinePayment]
payonline
paymentOnDelivery
publish (transaction)
consume (transaction)

1.1.3. Birds-eye view

Firstly, we want to give an overview on our bounded contexts included in the solution. For designing our system we thought about the eShop in a real world setting and came up with following relevant subdomains for our eShop domain:

- Authentication Subdomain
- User Subdomain
- Inventory Subdomain
- Product Subdomain
- Payment Transaction Subdomain
- Shipment Subdomain
- Notification Subdomain

In our design process we looked for a solution how the bounded contexts of our eShop would communicate to each other. An important step was to define what bounded context provides data for another bounded context. The diagram below gives an overview about the result of our design process from problem space to solution space and how the communication between the bounded contexts is established.


Problem Space
Solution Space
eShop Domain
User Subdomain
Authentication Subdomain
Inventory Subdomain
Payment Transaction Subdomain
Notification Subdomain
Shipment Subdomain
Design Process
eShop
Product Bounded Context
User Bounded Context
Authentication Bounded Context
Shared Kernel
Inventory Bounded Context
Payment Transaction Bounded Context
Notification Bounded Context
Shipment Bounded Context
U
D

Secondly, we want to give an overview of all the components involved in our solution. We narrowed our domains down as described above and tried to define microservices to properly represent our domains. Each of these services can be best investigated in our deployment diagram.

We provide following microservices:

- at.electro.shop.auth.service
- at.electro.shop.bff
- at.electro.shop.inventory.service
- at.electro.shop.notification.service
- at.electro.shop.shipment.service
- at.electro.shop.transaction.service
- at.electro.shop.user.service

Each of these services has its own Postgres database:

- postgres-auth
- postgres-inventory
- postgres-notification
- postgres-shipment
- postgres-transaction
- postgres-user

Except for the BFF, which only serves as a single unified API for the frontend.

Further we provide a UI to confirm successful integration:

- electro-shop-ui

We use following docker images to provide a means of communication between the Microservices in Publish/Subscribe way:

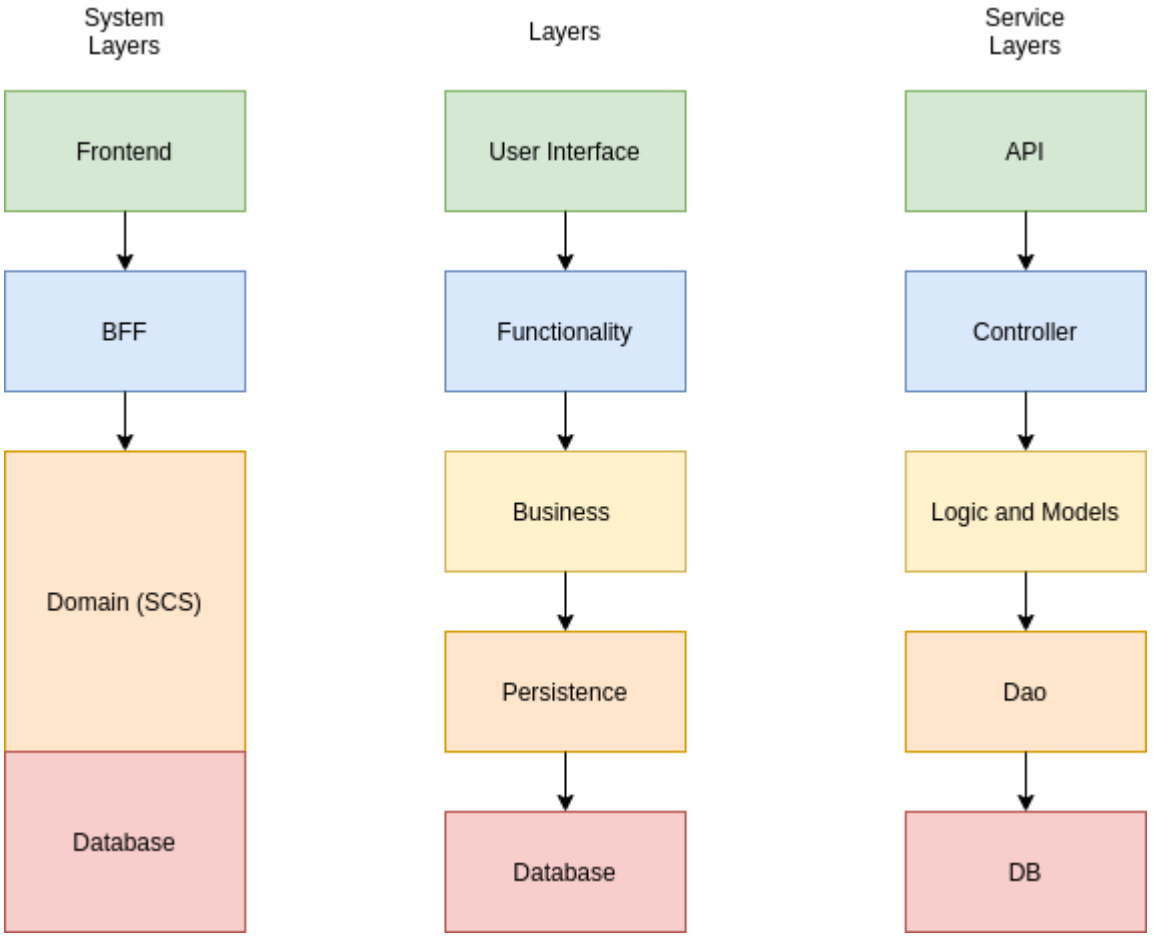
- zookeeper
- kafka
- kafdrop

Zookeeper came along with Kafka. It is a service registry to discover Kafka brokers and inform each broker about changes in the topology, so that information stays consistent between all the brokers in the network. We currently run only one instance of Kafka, as it is enough for the purpose of this lecture. But we could easily set up multiple instances in our configuration.

Kafdrop helps us to get an overview of the messages being sent between topics and heavily improves our debugging capabilities.

Layering

As described in the design report, we wanted to structure our solution into layers at the system level, same as the microservice level.



System Level

At the system level, the electro-shop-ui serves as our User Interface Layer.

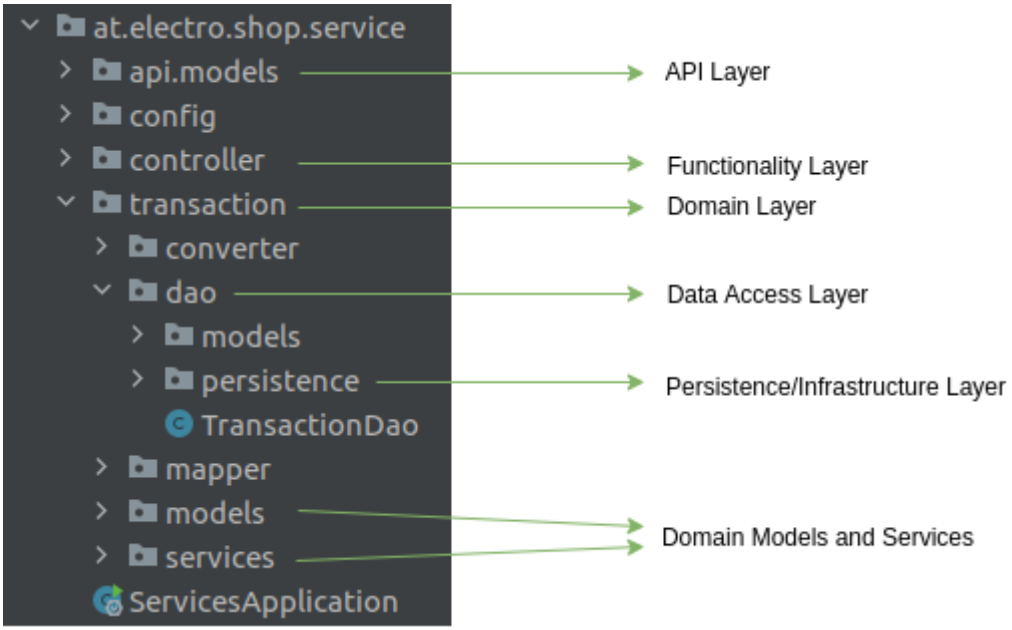
The at.electro.shop.bff serves as the functionality layer. It offers the UI a unified API with distinct operations that will operate later on our Microservices. On the system level, each of our Microservices, except for the BFF, represent certain Bounded Contexts. Each of them represent, on a very abstract level, our Domain Logic.

Therefore our microservices both implement our domain logic and store data to a database.

Lastly the databases represent the Infrastructure Layer on the System Level.

Microservice Level

The same pattern is applied to the individual microservices and was established before the implementation phase. This pattern was a guide for us on how to structure our packages to reflect the different layers we want to support.



This decision helped us very much. Agreeing to this structure assured that every Microservice is structured the same way. This increased readability, ensured that we could support each other and also provided a basic guideline for every member of the team on how to structure the code.

1.2. Architectural Design Decisions (ADDs)

NOTIFICATION SERVICE **In the context** of sending notifications to users for specific purposes, **facing** that duties are not adequately segregated, **we decided** to ensure that the notification logic is segregated from the logic querying products and updating shipments **to achieve** proper segregation of duties and ease future extending of new notification types **accepting** that the eventual sending of a notification may take longer and is not on the same service than the actual business logic.
LAYERING **In the context** of developing the individual microservices, **facing** the issue that we need to do many code reviews in microservices with different owners and the issue of properly organizing the code according to its responsibility **we decided** to structure the code according to distinct layers and wrote these layers down for everyone to follow **to achieve** well defined layers with distinct responsibilities aiming to provide better readability of the code, aiming also to have a consistent structure throughout all the microservices and therefore reducing the review and debugging time, while **accepting** that the implementation effort and testing effort slightly increases, because every layer also requires its own objects and in some cases, some layers only pass objects on or just map them.

PATH INSTEAD OF QUERY PARAMS IN TRANSACTIONS
In the context of retrieving the transaction history for sellers or venders or all
facing the fact that we need to somehow filter the transactions by different roles and that there were two options, either by addressing different transactions by query parameter or distinct paths
we decided to filter the transactions according to distinct paths. We added distinct paths for sellers, users and buyers to fetch the different histories for the different roles we support and only added query params to support pagination
to achieve proper separation of transaction histories and to add more clarity on what each request does. The advantages of different paths is that it is more understandable what each path tries to accomplish and hopefully confusion is reduced. Also verifying valid states on path params is slightly harder
accepting that we had to implement and support additional paths, just to accommodate different roles. Also accepting that this solution does not scale well, if we add additional roles to our application.

INTRODUCING A BFF
In the context of designing the application landscape
facing the issue to authenticate every request and also to make sure that the UI should only have one single source of information
we decided to implement an additional service, the BFF, to handle UI requests and check if requests are authenticated
to achieve a single source for the API we support, to reduce complexity for request authentication and token verification and to be able to only expose one service to the outside world
accepting that we have an additional service to support and to maintain and to deploy, which increases the implementation effort.

ADDING AN IN MEMORY DATABASE TO THE MICROSERVICES
In the context of implementing the individual microservices
facing the problem that we want to be able to implement our solutions independent from the complete setup, and independent from each other,
we decided to integrate an H2 Database in our Spring Boot applications, as to mock away the dependency for a running postgres database
to achieve independent development and rapid development. The developer is able to only start their individual microservices
accepting that we have the additional effort of integrating a new library and to create SQL queries that are compatible with both Databases. Also accepting the overhead of configuring two profiles.

SUPPORT MULTIPLE PROFILES
In the context of developing our microservices
facing the need of having different configurations for different scenarios
we decided to use the multi profile feature from Spring Boot
to achieve different configurations depending on the context and to achieve isolation of the microservice
accepting that we have to maintain multiple configurations, which incurs additional maintenance effort and increases complexity.

INVENTORY SERVICE
In the context of managing products in inventories of respective vendors,

facing the issue that the products are tightly coupled to the inventories, **we decided** that the inventory and product logic will be both located within a single Microservice, sharing a common database, **to achieve** seamless interchange among the two entities and faster API response, avoiding unnecessarily chatty APIs and database redundancy, while **accepting** the risk that both products and inventories of vendors may become unavailable in case of outage of the service.
SHIPMENT SERVICE **In the context** of updating the shipment status of purchased products, **facing** the fact that the eShop aims for a local distribution, **we decided** that vendors can manually update the status from the vendor UI, **to achieve** that small businesses without automated logistic centres can inform their customers about the shipment status of purchased products, **accepting** that vendors may forget about updating the shipment status.
USER SERVICE **In the context** of storing the products in the cart and able to buy the product from the cart **facing** the problem of having authorization and user database together **we decided** to isolate the concerns for the authorization and activity of users **to achieve** the separation of domain for the authorization and user **accepting** that the authorization service is not related to the storage of user's others information rather than only its authority or role in the system.

1.3. Major Changes Compared to DESIGN

Changes in system design architecture in diagrams:

- Compared to the system design for DESIGN submission, the diagrams now include bounded contexts.
- The diagrams of the 4+1 Views Model were updated according to the feedback we got on the DESIGN submission. The Scenarios View now includes two additional diagrams (Authentication and Administration) for a more detailed view. The Physical View now displays a detailed overview of the physical nodes of the system and the artefacts that are being deployed.

Changes in microservice architecture:

- Swagger: We now document our endpoints via Swagger. They are automatically generated when we do bff tests.
- The price suggestion provided by the web crawler is not located in the Inventory Microservice.
- The user name is the email of the user and not an additional user name.
- Transaction Microservice:
    - The history by query parameters was changed to its own path for each transaction.

    - The microservice publishes a list of transactions instead of a single transaction.
    - Request is a single user transaction.
    - For every product in the request there is a row in the transaction database.
    - Each entry created by a single product is collected in a list that is published to the event store → so we publish a list of transactions
    - Added pagination possibilities to restrict the amount of results by default
- Inventory Microservice:
    - The Kafka topic “product.updated” is now only published for updated prices and not for other types of product updates because it only affects the price alert for products handled by the Notification Microservice.
- Shipment Microservice:
    - Shipment MS receives a list of transactions with one product each. Every product is consequently handled as a single shipment.
- User Microservice:
    - Cart will not have just a list of strings for product id, rather it will contain the product list itself but with only id in it.
- Notification Microservice:
    - The notification service now includes a Template builder. This enables us to create better email templates and put in more useful information for the user
    - The Email Model has a list of receivers instead of a string. With this, we can send the email to more users at once. We can add multiple receivers for administration purposes in the application properties.
    - The notification service now also provides interfaces for getting notifications per user and deleting notifications.

1.4. Development Stack and Technology Stack

1.4.1. Development Stack

- Version control system: GitLab
- Create Build Pipeline with Gitlab CI
    - Build Steps will be described in the .gitlab-ci.yml
        - (Optional) Pre-build static code analysis
        - Build applications
        - Execute Tests
        - (Optional) Quality Gate
        - Create Docker Image
        - Deploy Image
- Build system:
    - Front-End: Yarn
    - Back-End: Maven

    - For the deliverable image: Docker (Docker Compose)

1.4.2. Technology Stack

- Programming languages:
    - Back-end: Java
    - Front-end: Javascript
- Frameworks:
    - SpringBoot
    - React.js
        - react-router
        - styled-components
        - Material Design
- Test-Frameworks:
    - Junit5
    - Rest-assured
    - (Optional) Cypress
- Databases:
    - PostgreSQL
    - Flyway for database migration
    - H2 Database (Testing)
- Servers:
    - The back-end microservices will be running in embedded tomcat servers, provided by the SpringBoot framework
- Containers:
    - Docker
    - Docker Compose is used to handle and manage our multi-container application
- Event Store or Pub/Sub:
    - Kafka + Zookeeper + Kafdrop
- IDE:
    - Front-End: JetBrains Webstorm
    - Back-End: JetBrains Intellij
- Libraries:
    - FasterXML
- Mocking:
    - Mockable
    - Postman

2. Updated Requirements

PROVIDE AN UPDATED VERSION OF YOUR DESIGN SUBMISSION CONTENT:
Updated requirements of the system, their implications on the implementation (e.g. limitations, constraints), and how you verified the requirements (e.g., test cases, manual inspection procedure). Include the status of the verification: is the implementation completely meeting the requirements?

Consider the characteristics of good requirements[1] and decide on a set of properties (e.g., identifier, priority, description, affected use-cases and user roles, verification method) to define a requirement well and how you intend to verify it (e.g. test cases, manual inspection/review). List and discuss the identified main functional and non-functional requirements of the system and their implications on the implementation (e.g. limitations, constraints).
Regarding grading we expect approximately 3 requirements per team member.

Integration tests executed on every build
Verified against the diagrams (component, use case) and made changes to the diagram and the implementation (iteratively)

Final user test, verify if requirements fulfill user expectations

ID	Description	Implication on implementation	Verification
Shipment	A user receives an email when their shipment status changes	We have a notification service that listens to changes of a shipment update and sends notification emails to the users.	We set up a test mailbox that receives shipment update emails.
Product marking	A user can mark a product to receive a price alert when the price of a product falls to a benchmark.	We have a notification service that listens to changes of product prices and sends price alert notification emails to the users.	We set up a test mailbox that receives price alert emails.
Usability	Our eShop must be intuitive for users. Users need to be able to navigate the eShop	We need to uncover usability issues.	UX testing with users.
Availability	Our shop must offer interested customers a certain amount of up-time. Therefore the	Therefore, we must build our system to be scalable. We will use docker to package our	Monitoring tool, that will track the amount of transactions being

[1] https://en.wikipedia.org/wiki/Requirement#Characteristics_of_good_requirements

	application must be robust. Because at the beginning, we are only a regional e-Shop, we aim to reach two nines (99%) of availability.	applications in containers and we intend to manage those containers within a container orchestration system. We will provide at first, two instances of each of our services, so if we were to update any of them, the other can handle user requests. We will also use a load balancer, to distribute load according to availability and capacity of each service. Details in the Deployment Design.	served compared to the transactions that we had to decline.
Scalability	Our shop intends to grow, therefore we want to be able to handle a growing number of user requests.	Our system must be separated into individual, scalable containers, these will be managed by a container orchestration system. To handle additional services, we will employ a load balancer, and a service registry.	We will design system tests that will simulate load spikes, as they would occur during high traffic times or as they would appear in a projected growth scenario.
Security	We are dealing with sensitive user data, therefore our business should make sure to secure that data carefully.	We make sure that we support user passwords of length up to 64 characters. Long passwords take longer to be cracked and offer therefore a higher security. To ensure that 64 characters are supported, we use bycrypt encryption. It supports 64 characters and it makes sure that we never store the passwords as is, but just an encrypted representation of it. All requests being made to our environment, need to pass through the	During development, we used a security checklist, containing the most important dos and don'ts. Each tester is provided with a similar security checklist and performs regular checks. We also try to make use of automatic security scanning tools, such as SonarQube for Security Flaws in our Codebase.

		BFF. The BFFs is the main authority to check user requests for a valid authenticated token. We use a server side authentication mechanism, offered by Spring Boot security. and a login page rendered server side.	Pre-Release we employ a security company to perform an intrusion and vulnerability test against our service.
Maintainability	Our services are easy to maintain. Meaning they are easy to be updated and repaired, if a service fails.	We will focus on logging, so we can easier identify errors. The code is split into different microservices, so if one fails, the other ones are not affected. We commit to a coding standard and software designs across every microservice, to increase code readability, making it easier for other people to maintain every service. Also unit tests and integration tests make sure that our code is modular and tested.	We measure our unit test coverage and run the code through a linter and SonarQube, to verify that coding standards are applied. We provide integration tests to the integrated functionality of the system.
Reliability	Our services need to be reliable by providing consistent performance.	We will use a Load Balancer in our system and run multiple instances for every service, just as described in the deployment diagram. This will ensure that load is evenly distributed and therefore increases the reliability of the system.	We will run performance and stress tests on our system. We will also implement a monitoring tool, in which we can see the load compared to response times and resources of the system.
Functional Requirements	As described by the use cases	See use cases	See use cases
System Configurati	The system should be easy configurable, and	We use Docker in our microservices, which	System can be run on different hosts

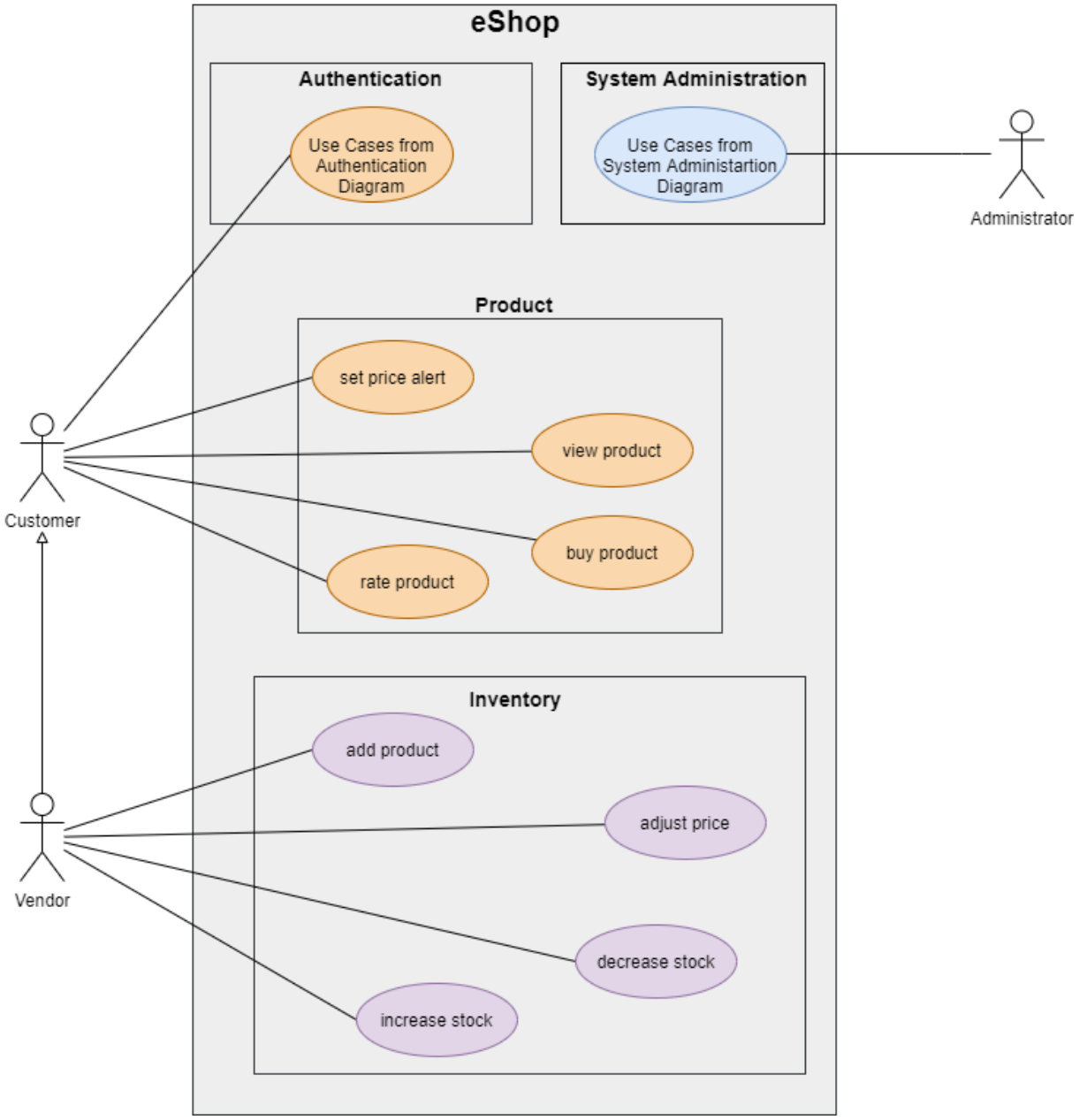
on Management	run on multiple machines	allows them to be independent of the hosts system. It also serves as a central point for configuring the different containers.	without extra configuration steps for the host system
Processing Time	The system should respond in an appropriate time window, so users do not complain about bad usability	The load balancer, combined with the multiple instances of services and separation of concerns for services, will guarantee a good overall system performance.	Performance tests will be done regularly, to keep it steady. New features will go through a performance testing phase.
Data Architecture	Data needs to be separated by Domains	We define microservices for each bounded context and necessary data is stored in a database specifically for that microservice.	We iteratively changed the diagrams and the code to implement the bounded contexts.

3. Updated 4+1 Views Model

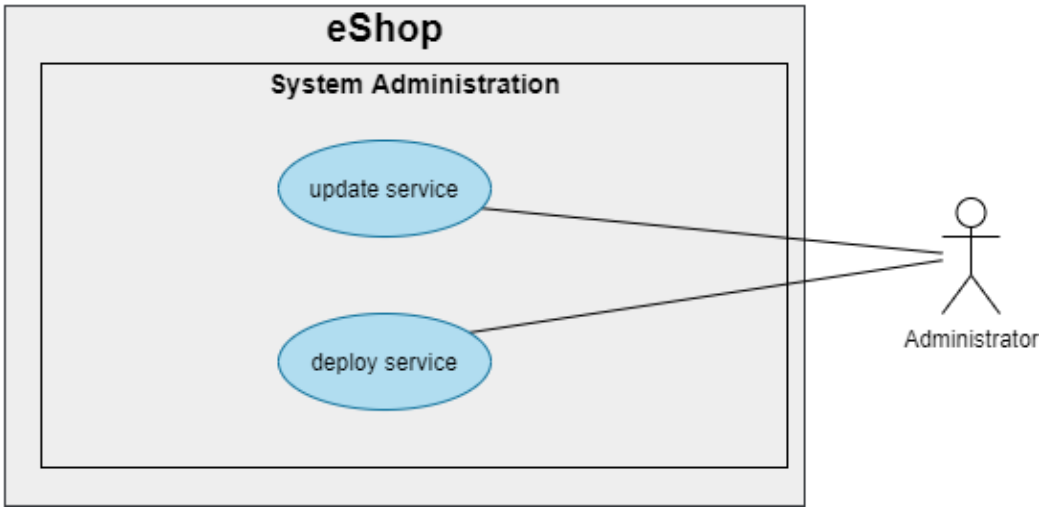
3.1. Scenarios / Use Case View

3.1.1. Use Case Diagram(s)

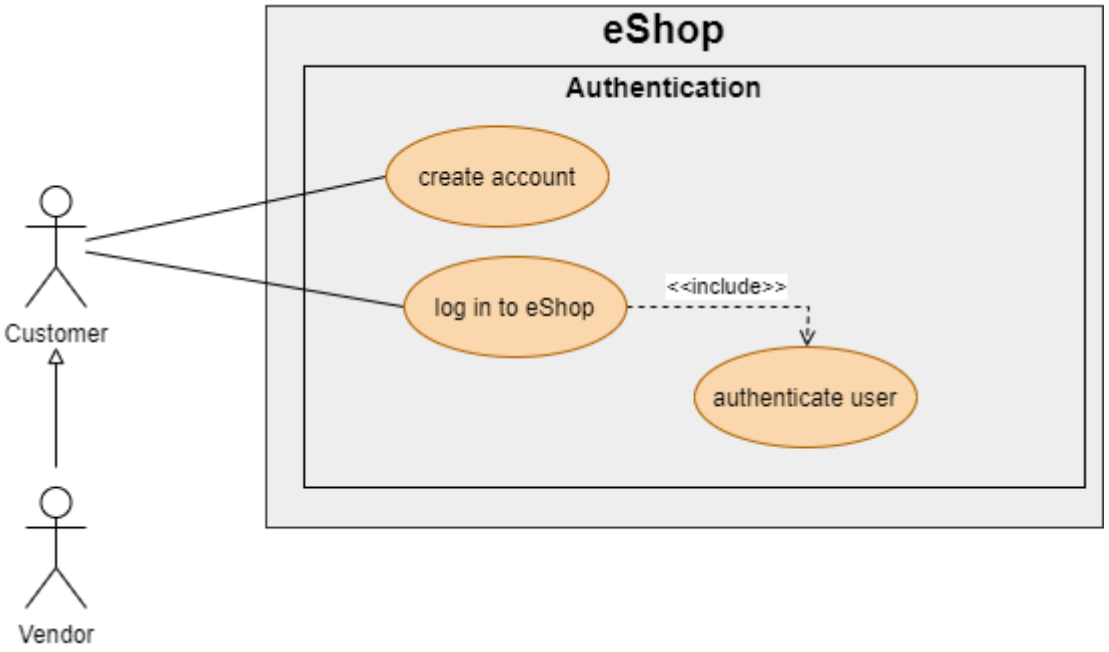
3.1.1.1 General eShop Use Case Diagram



3.1.1.2 eShop Authentication Diagram



3.1.1.3 eShop Authentication Diagram



3.1.2. Use Case Descriptions

eShop Authentication Diagram

Use case:	Create account
Use case ID:	UC 01
Actor(s):	Customer
Brief description:	The Customer creates a user account for the eShop.
Pre-conditions:	The Customer visits the eShop's website.

Post-conditions:	The Customer uses the user account to log in to the eShop.
Main success scenario:	1. The Customer selects “create account”. 2. The Customer is redirected to the registration page. 3. The Customer fills in their username and selects a password. 4. The Customer submits the registration form. 5. The Customer receives a verification email and has to verify their email address by clicking on a link. 6. The Customer will be forwarded to the store front and has successfully created a user account.
Extensions:	At every point, a user can opt out.
Priority:	Low
Performance target:	Response time should be less than 50 ms. The system should be able to process 5000 registrations at one point in time.
Issues:	

eShop Authentication Diagram

Use case:	Log in to eShop
Use case ID:	UC 02
Actor(s):	Customer
Brief description:	The Customer logs in to the eShop.
Pre-conditions:	1. The Customer visits the eShop’s website. 2. The Customer has already created a user account.
Post-conditions:	The Customer has access to the eShop’s areas that require a user account.
Main success scenario:	1. The Customer selects “log in”. 2. The Customer is redirected to the login page. 3. The Customer fills in username and password. 4. The Customer submits the login form. 5. The user is authenticated in a way that the provided username and password are validated. 6. The Customer is successfully logged in to the eShop.

Extensions:	The Customer is automatically logged out when the session expires.
Priority:	High
Performance target:	Response time should be less than 50 ms.
Issues:	

Use case:	Set price alert
Use case ID:	UC 03
Actor(s):	Customer
Brief description:	The Customer sets a price alert, i.e. sets an alarm to get notified when the price of a monitored product passes a threshold.
Pre-conditions:	1. The Customer is logged in. 2. The Customer views a product.
Post-conditions:	The Customer gets notified when the price of the monitored product passes a threshold.
Main success scenario:	1. The Customer selects “set price alert” on the product page. 2. The Customer sets a threshold for the alert. 3. The Customer saves the entry.
Extensions:	The eShop sends an email to the Customer when the price of a monitored product passes the selected threshold.
Priority:	Low
Performance target:	
Issues:	

Use case:	View product
Use case ID:	UC 04
Actor(s):	Customer
Brief description:	The Customer views a product in the eShop.
Pre-conditions:	● The Customer visits the eShop’s website. ● Products need to be in the inventory.
Post-conditions:	The Customer closes the product page.
Main success scenario:	1. The Customer selects a product on the list of available products. 2. The Customer is redirected to the product page. 3. The available product information is displayed.
Extensions:	● The Customer can navigate back to the product list and view another product.

	● The Customer sees product suggestions of the same product category.
Priority:	High
Performance target:	Response time should be less than 100 ms.
Issues:	

Use case:	Buy product
Use case ID:	UC 05
Actor(s):	Customer
Brief description:	The Customer buys a product.
Pre-conditions:	1. The Customer is logged in. 2. The Customer views a product.
Post-conditions:	The product is shipped.
Main success scenario:	1. The Customer clicks on “add to shopping cart”. 2. The Customer clicks on “buy product”. 3. The Customer chooses the shipping address. 4. The Customer chooses the payment form. 5. The Customer clicks on “finish order process”. 6. The Customer will get a notification that a product was bought. 7. The Customer will get a notification that a product is out for delivery. 8. The Customer will get a notification that a product was delivered.
Extensions:	The Customer can rate the purchased product.
Priority:	Medium
Performance target:	The system should be able to handle 10 000 transactions at once.
Issues:	

Use case:	Rate product
Use case ID:	UC 06
Actor(s):	Customer
Brief description:	The Customer rates a purchased product.
Pre-conditions:	1. The Customer is logged in. 2. The Customer views a product.
Post-conditions:	The Customer receives a confirmation of having rated the product.

Main success scenario:	1. The Customer clicks on “rate product”. 2. The Customer rates the product with stars. 3. The Customer submits the rating form.
Extensions:	
Priority:	Low
Performance target:	
Issues:	

Use case:	Add product
Use case ID:	UC 07
Actor(s):	Vendor
Brief description:	The Vendor adds a product to the inventory.
Pre-conditions:	● Vendor accounts need to be in the database. ● The Vendor is logged in.
Post-conditions:	The product appears in the eShop.
Main success scenario:	1. The Vendor clicks on “add product”. 2. The Vendor fills in the required information for adding a new product. 3. The new product is successfully added to the Vendor’s inventory.
Extensions:	● The Customer can buy the added product. ● The Vendor can adjust the price and update the current stock of the product. Note: Added products cannot be deleted to preserve the product ID for future product related processes (e.g. user requests).
Priority:	Low
Performance target:	Response time should be less than 50 ms
Issues:	

Use case:	Adjust price
Use case ID:	UC 08
Actor(s):	Vendor
Brief description:	The Vendor adjusts the price of a product in the inventory.
Pre-conditions:	● The Vendor is logged in. ● The Vendor must have already added some products.
Post-conditions:	The product is displayed with the adjusted price.
Main success scenario:	1. The Vendor clicks on “adjust price”.

	2. The Vendor receives a suggested price based on the result of a web crawler. 3. The Vendor accepts the suggested price or chooses to select a different price. 4. The Vendor saves the updated product information. 5. The price is successfully updated.
Extensions:	-
Priority:	Medium
Performance target:	Web crawler price suggestion should not take longer than 2 sec (suggested time by google).
Issues:	

Use case:	Decrease stock
Use case ID:	UC 09
Actor(s):	Vendor
Brief description:	The Vendor decreases the availability of an existing product in the inventory.
Pre-conditions:	● The Vendor is logged in. ● The product stock is owned by the Vendor. ● The product stock is available (was added). ● The product stock availability is more than 0.
Post-conditions:	The product is displayed with the updated availability.
Main success scenario:	1. The Vendor can call the “decrease stock” operation. 2. The Vendor sets the decreased number of available products. 3. The Vendor confirms the save of the updated product information. 4. The availability of the product is successfully decreased.
Extensions:	If the availability is set to zero, in the eShop the product is listed as out of stock.
Priority:	Medium
Performance target:	Updated availability on the front-end will be refreshed less than 1 minutes after the database propagation.
Issues:	

Use case:	Increase stock
Use case ID:	UC 10

Actor(s):	Vendor
Brief description:	The Vendor increases the availability of a created product in the inventory.
Pre-conditions:	The Vendor is logged in. The product stock is owned by the Vendor. The product stock is available (was created).
Post-conditions:	The product is displayed with the updated availability.
Main success scenario:	1. The Vendor calls "increase stock" operation. 2. The Vendor sets the increased number of available products. 3. The Vendor saves the updated product information. 4. The availability of the product is successfully increased.
Extensions:	Stock is bounded (cannot be increased over a certain limit).
Priority:	Medium
Performance target:	10000 Vendors should be able to increase the stock at the same time
Issues:	

eShop System Administration Diagram

Use case:	Update service
Use case ID:	UC 11
Actor(s):	Administrator
Brief description:	The Administrator maintains the system by updating services in the eShop.
Pre-conditions:	1. New code for a microservice is available. 2. The new code was tested. 3. The Administrator is logged in to GitLab.
Post-conditions:	No unwanted side effects occur on the eShop after the deployment.
Main success scenario:	The deployment is successfully completed.
Extensions:	A roll back of the deployment is possible.
Priority:	High.
Performance target:	
Issues:	

eShop System Administration Diagram

Use case:	Deploy service
Use case ID:	UC 12
Actor(s):	Administrator

Brief description:	The Administrator can deploy new services to the eShop.
Pre-conditions:	4. A new service is available. 5. The new code was tested. 6. The Administrator is logged in to GitLab.
Post-conditions:	No unwanted side effects occur on the eShop after the deployment.
Main success scenario:	The deployment is successfully completed.
Extensions:	A roll back of the deployment is possible.
Priority:	High.
Performance target:	
Issues:	

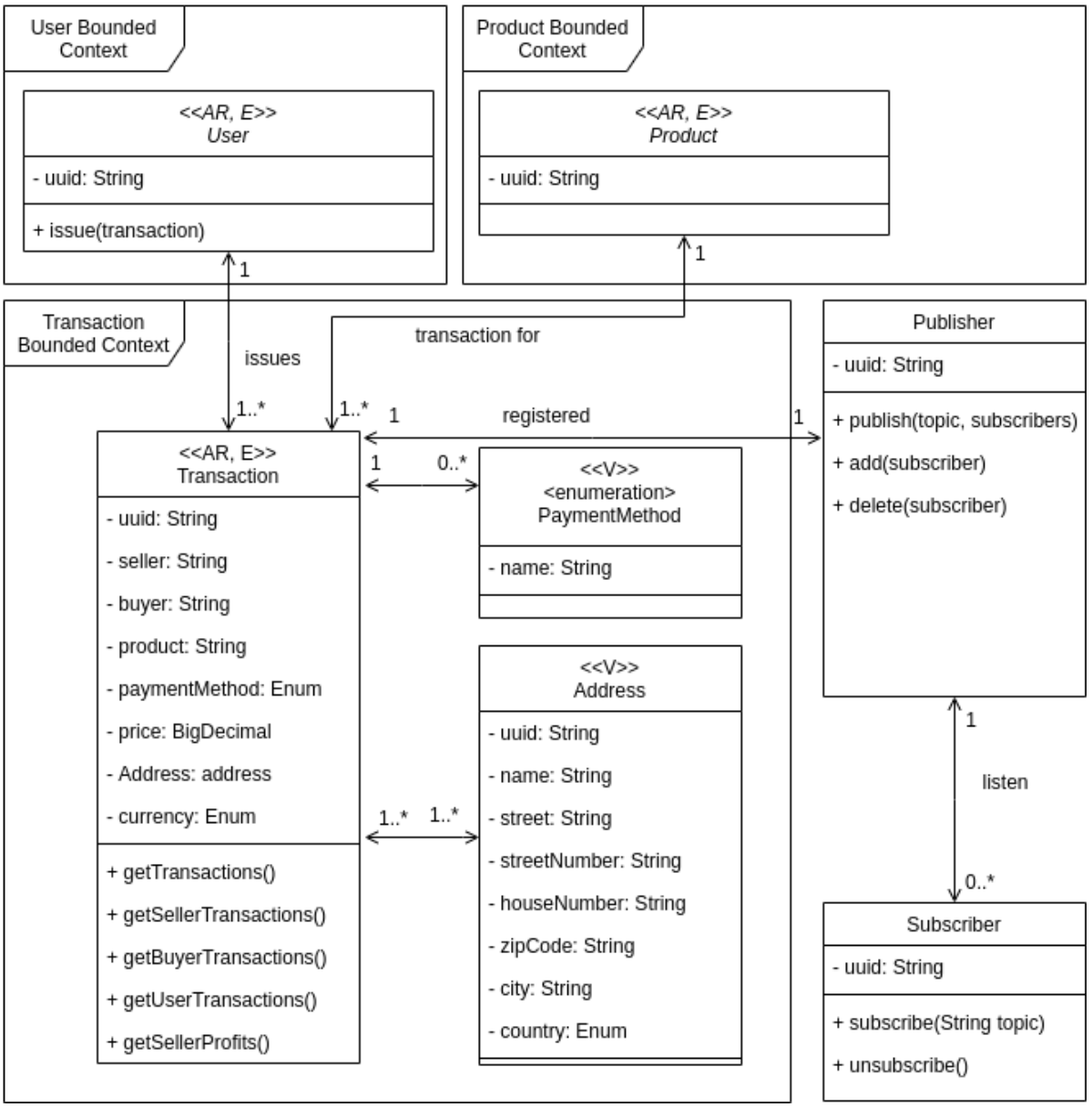
3.2. Logical View

3.2.1. Payment Transaction Bounded Context

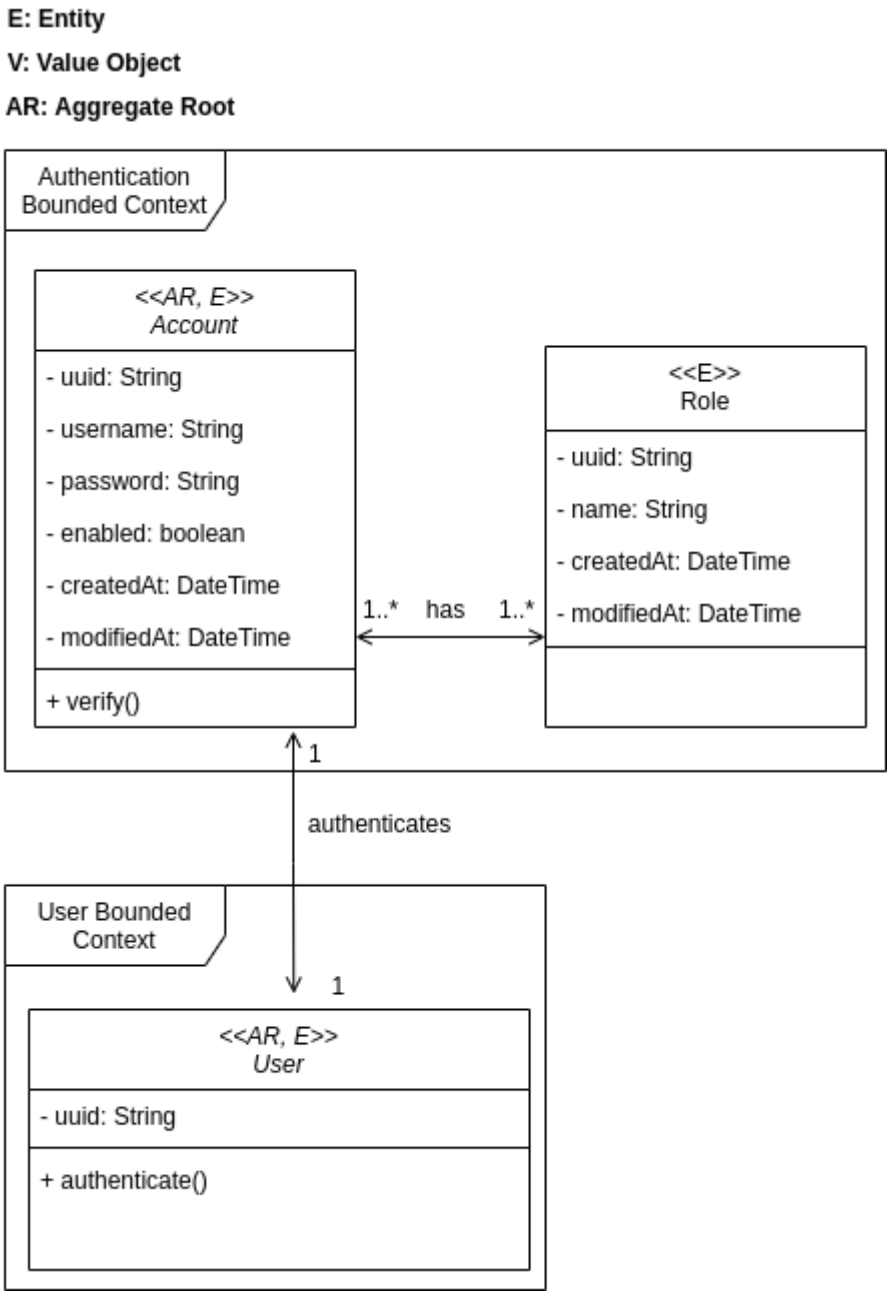
E: Entity

V: Value Object

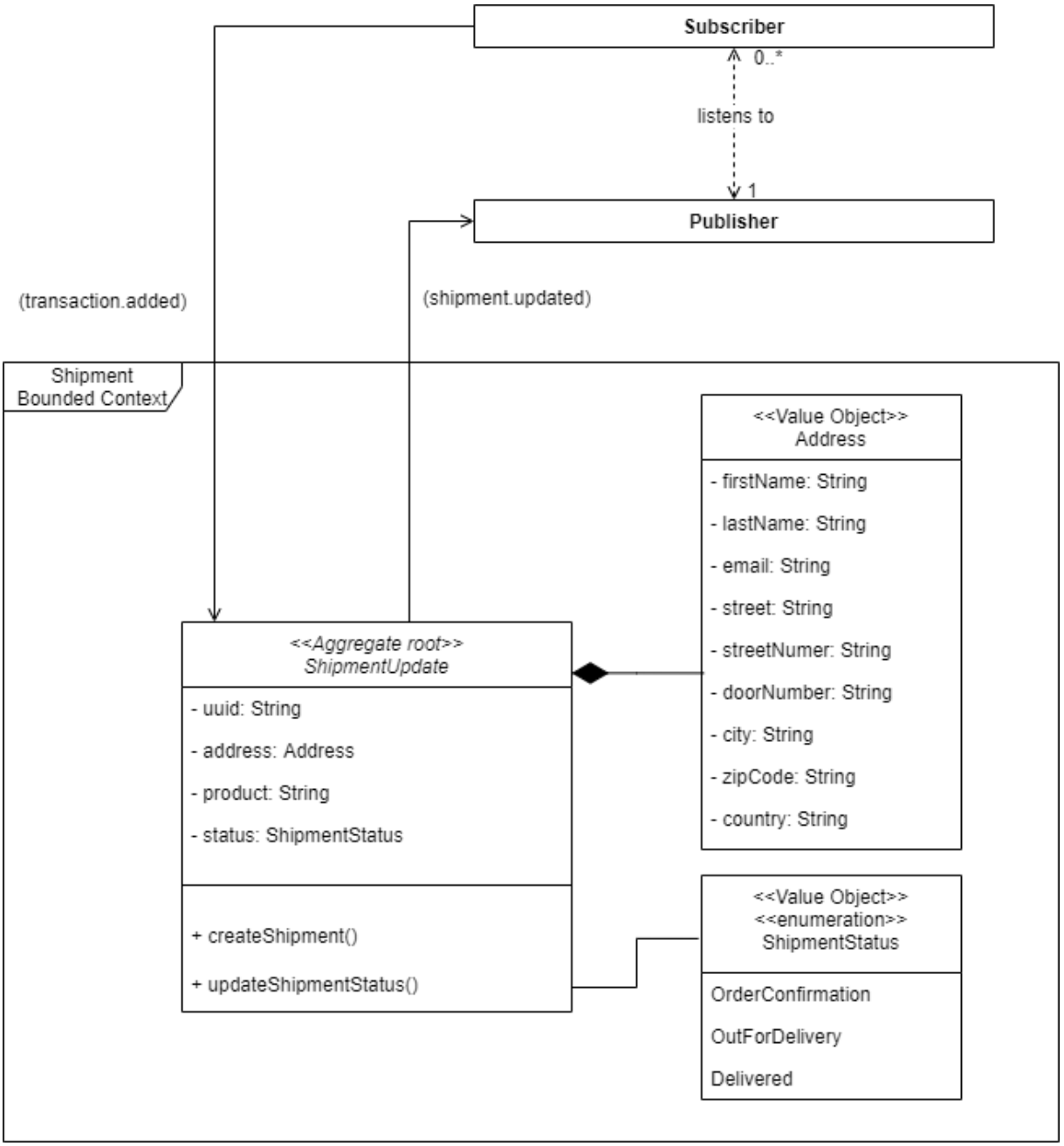
AR: Aggregate Root



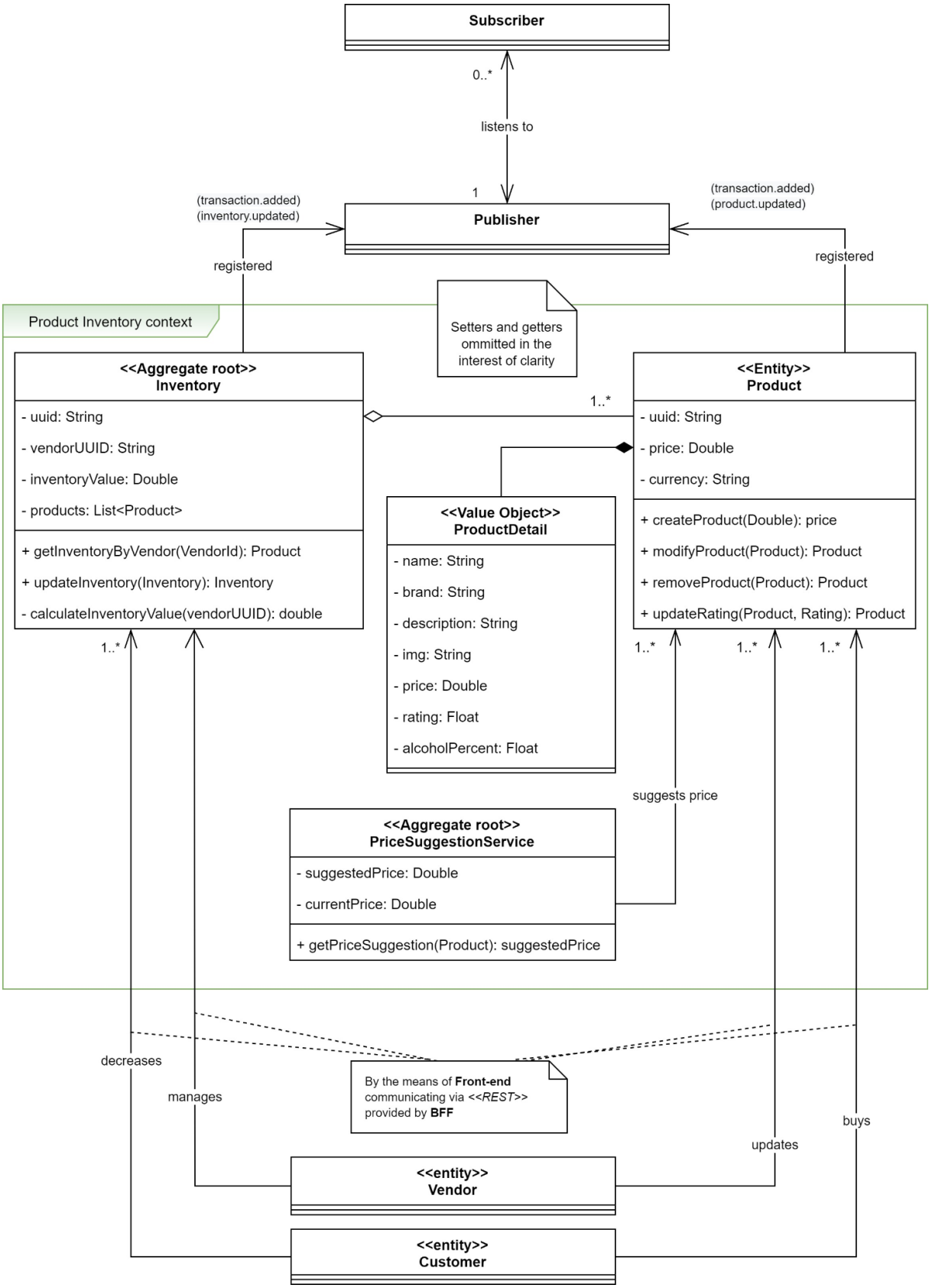
3.2.2. Authentication Bounded Context



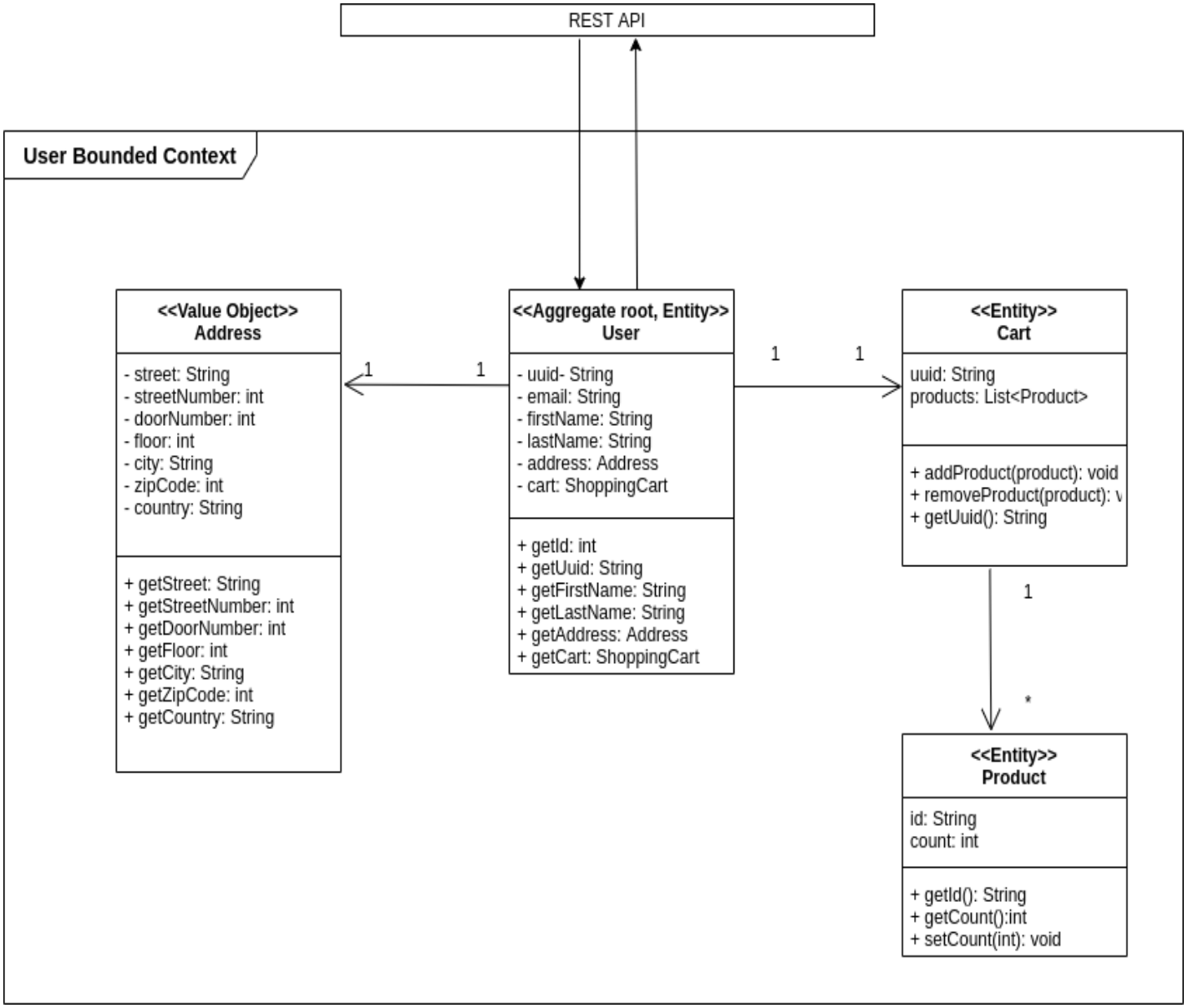
3.2.3. Shipment Bounded Context

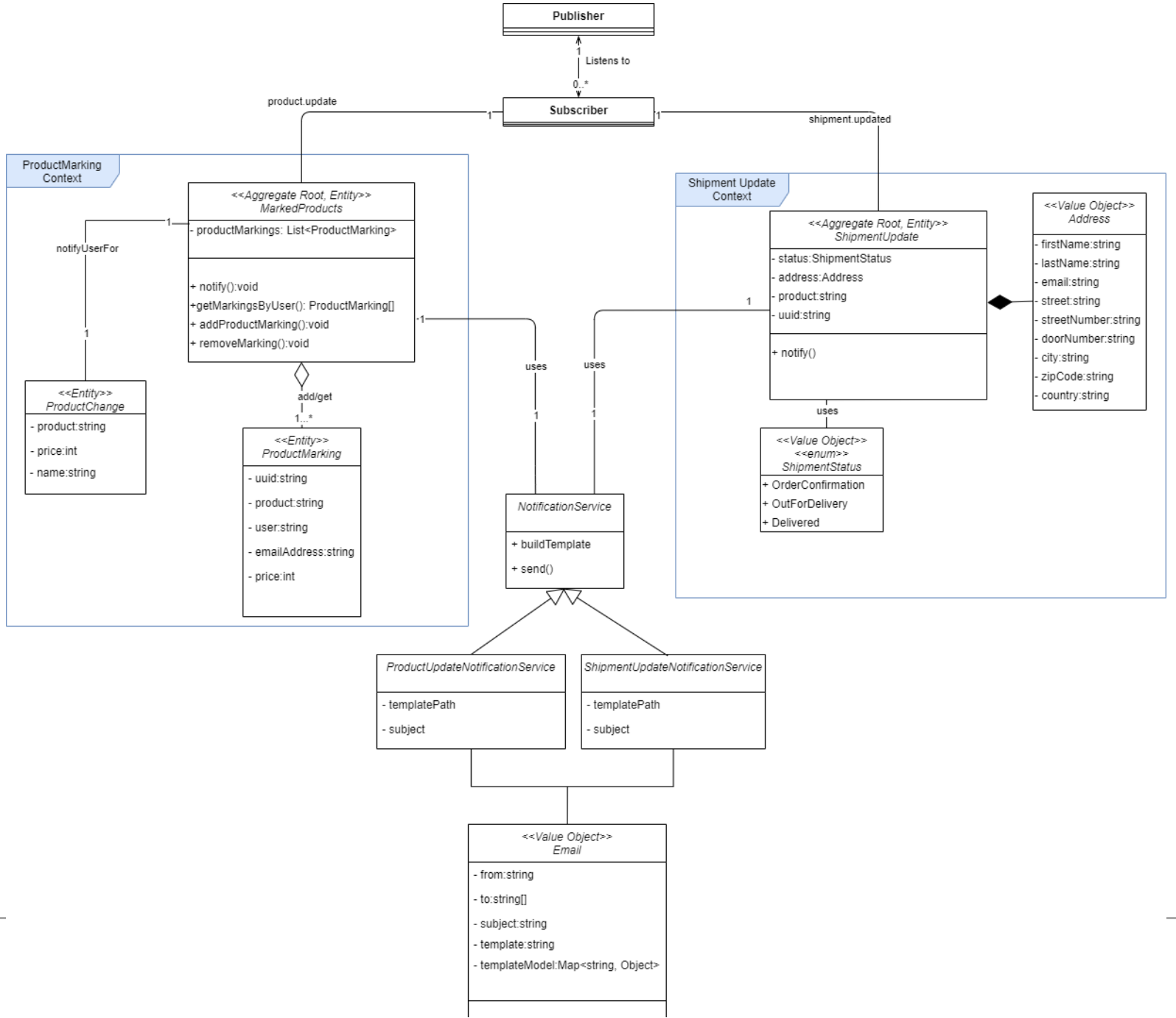


3.2.4. Product Inventory Context



3.2.5. User Microservice




Publisher
1
Listens to
0..*
Subscriber
product.update
1
1
shipment.updated
ProductMarking Context
<<Aggregate Root, Entity>>
MarkedProducts
- productMarkings: List<ProductMarking>
+ notify():void
+getMarkingsByUser(): ProductMarking[]
+ addProductMarking():void
+ removeMarking():void
1
notifyUserFor
1
<<Entity>>
ProductChange
- product:string
- price:int
- name:string
add/get
1...*
<<Entity>>
ProductMarking
- uuid:string
- product:string
- user:string
- emailAddress:string
- price:int
1
uses
1
uses
Shipment Update Context
<<Aggregate Root, Entity>>
ShipmentUpdate
- status:ShipmentStatus
- address:Address
- product:string
- uuid:string
+ notify()
1
<<Value Object>>
Address
- firstName:string
- lastName:string
- email:string
- street:string
- streetNumber:string
- doorNumber:string
- city:string
- zipCode:string
- country:string
uses
<<Value Object>>
<<enum>>
ShipmentStatus
+ OrderConfirmation
+ OutForDelivery
+ Delivered
NotificationService
+ buildTemplate
+ send()
ProductUpdateNotificationService
- templatePath
- subject
ShipmentUpdateNotificationService
- templatePath
- subject
<<Value Object>>
Email
- from:string
- to:string[]
- subject:string
- template:string
- templateModel:Map<string, Object>

3.3. Process View

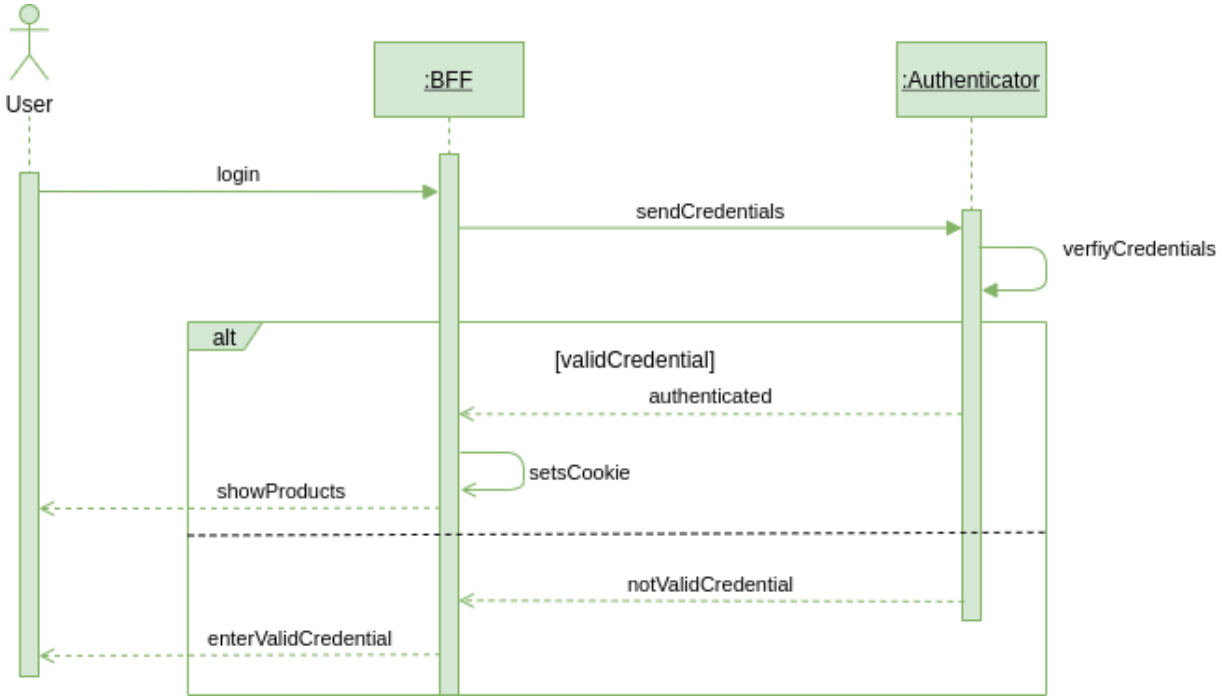


FIG 1: USER LOGIN

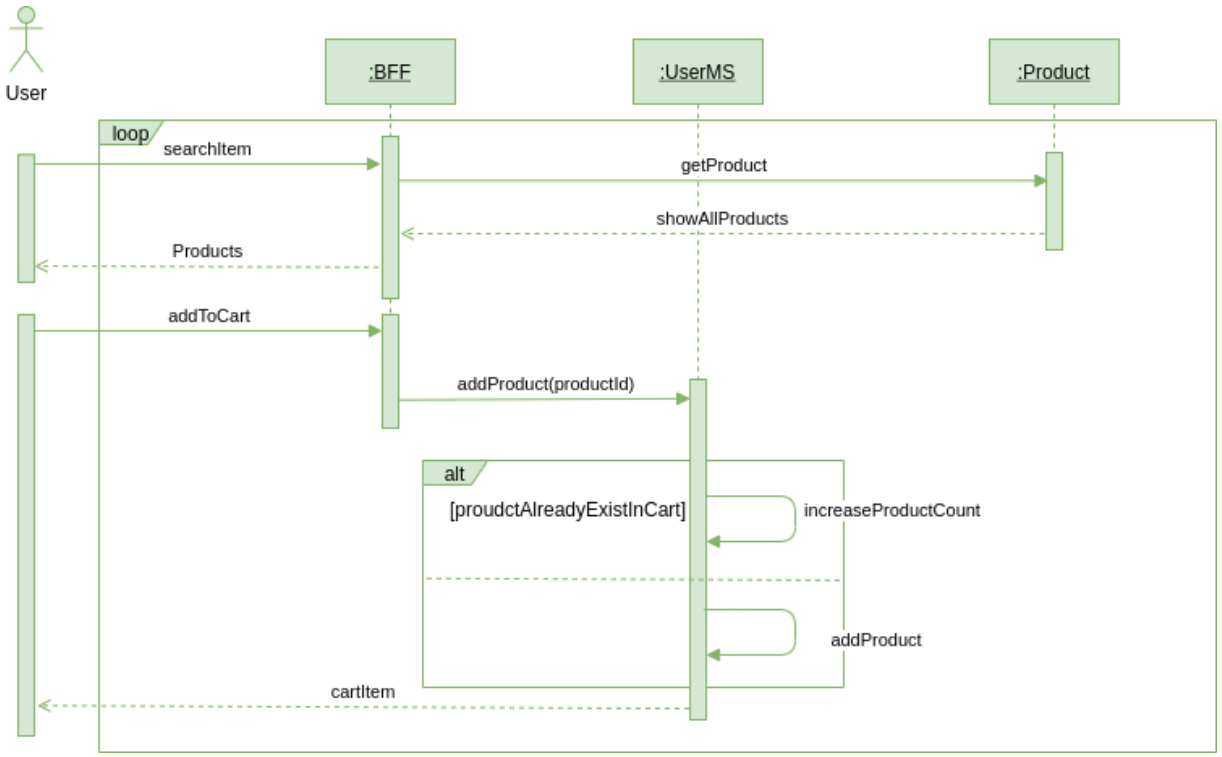


FIG 2: USER ADD PRODUCT TO CART

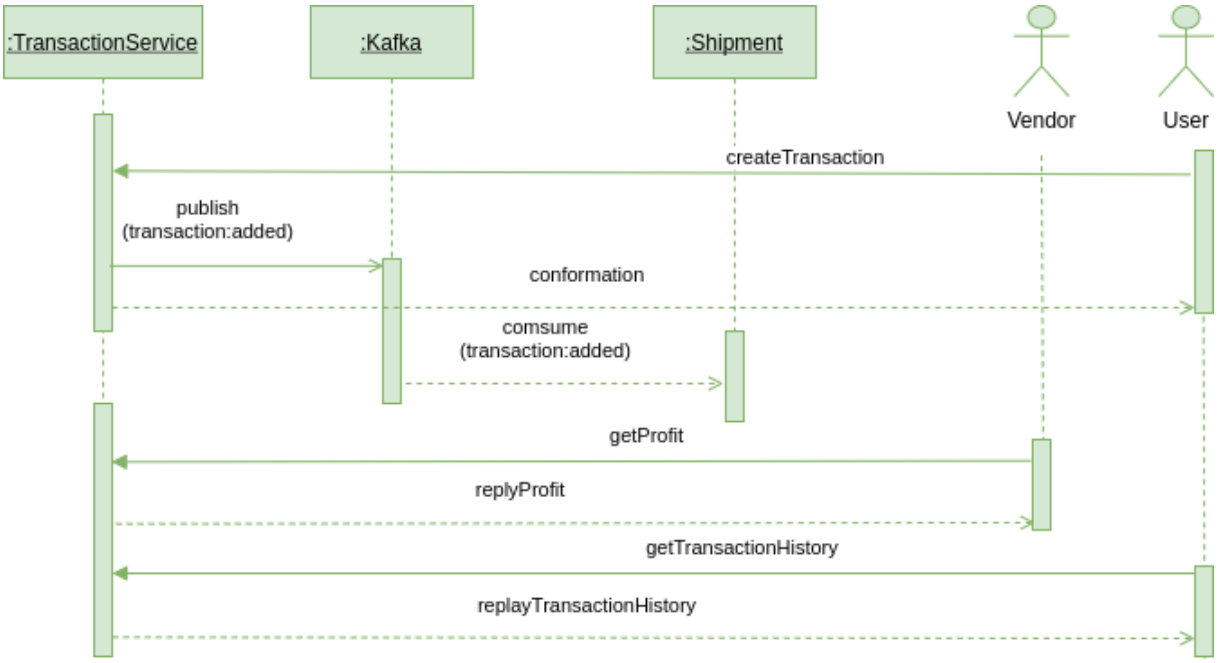


FIG 3: TRANSACTION & VIEW HISTORY

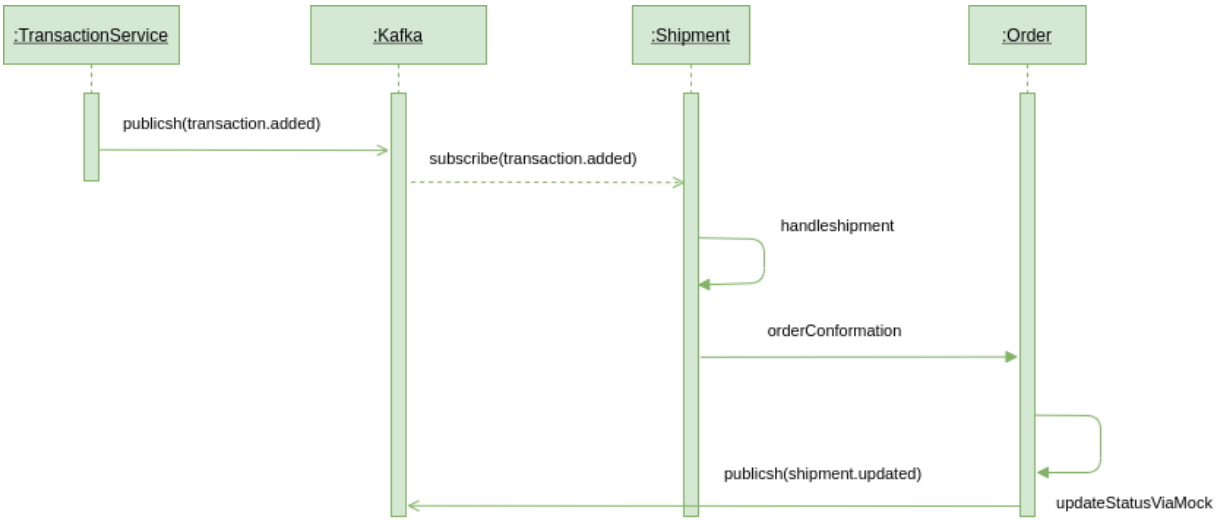


FIG 4: TRANSACTION SHIPMENT

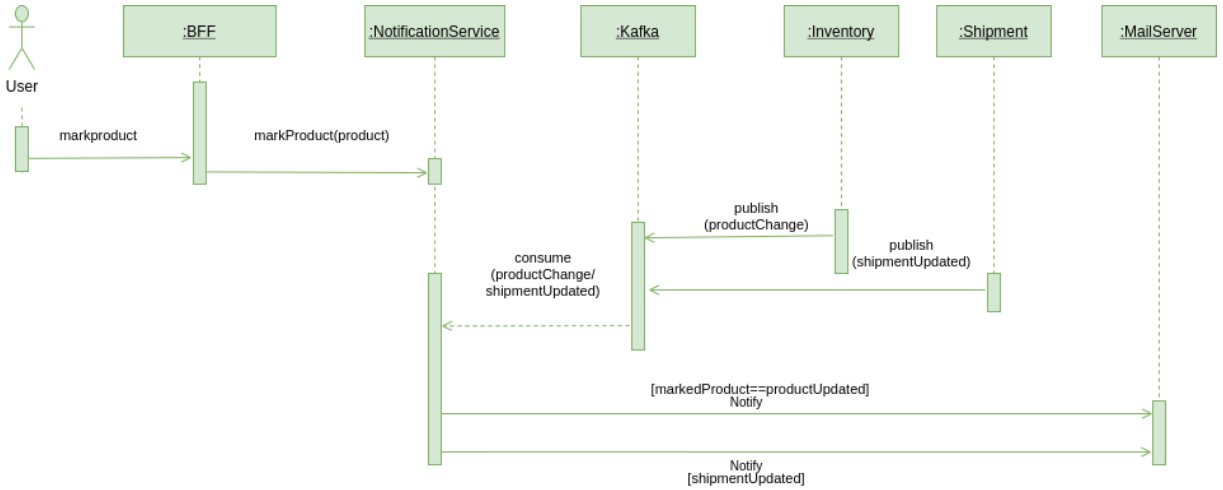


FIG 5: NOTIFICATION TO USER ABOUT SHIPMENT AND MARKED PRODUCT

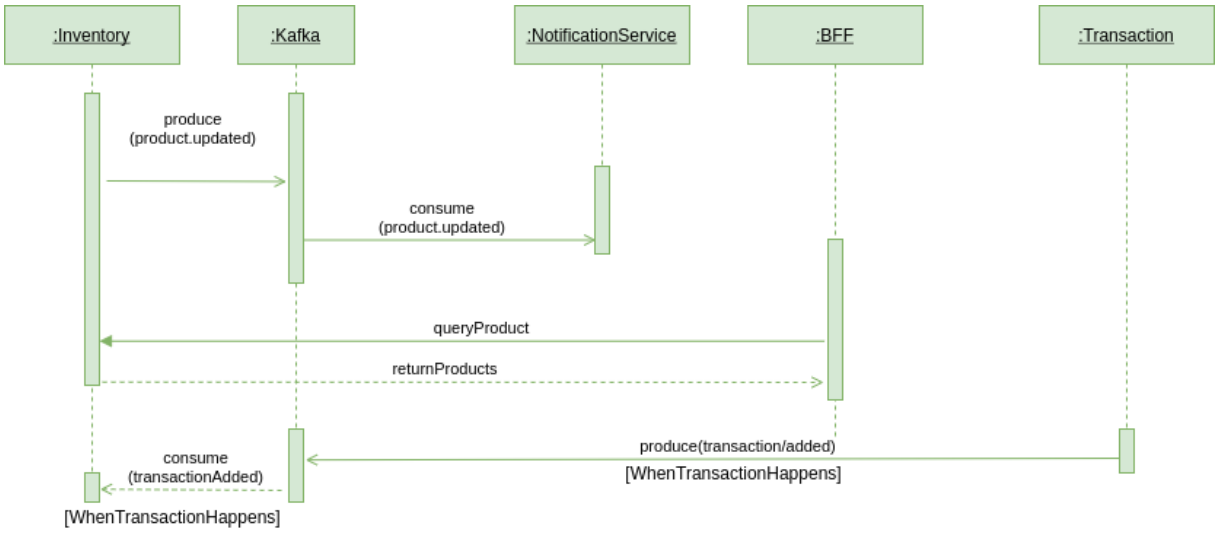
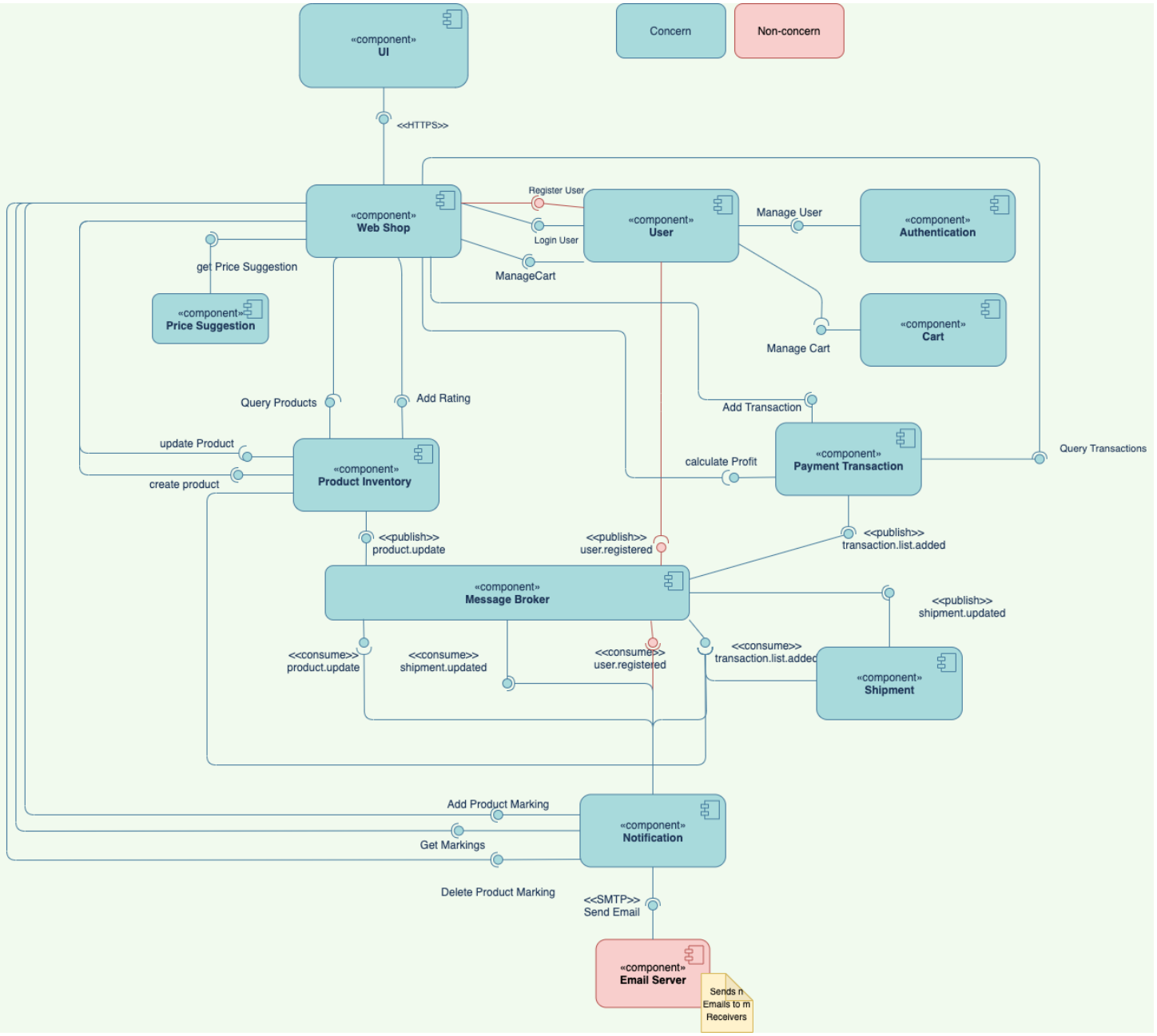


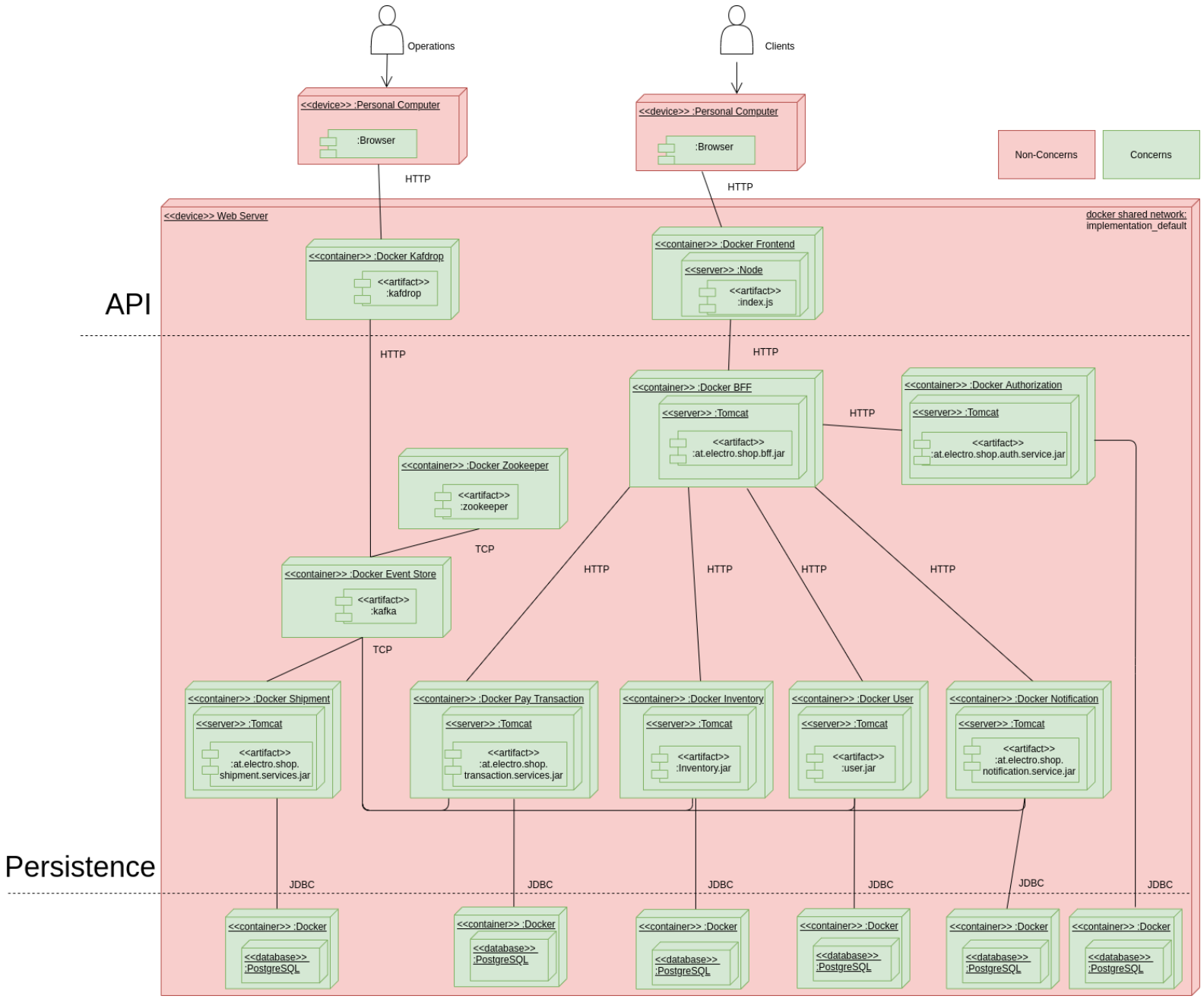
FIG 6: INVENTORY GETTING NOTIFICATION OF SHIPMENT

3.4. Development View



3.5. Physical View

Deployment Diagram



4. Continuous Delivery

Our application supports several means of building, testing and deploying an application.

At any moment, a developer is able to perform a full build of all artifacts, execute all tests and deploy it to the deployment folder. For these tasks we provided several shell scripts:

- **startup_local.sh** → will issue a full build, run tests and startup the environment
- **deploy.sh** → will fetch all docker images, create a tar file and deploy it to the deployment folder
- **build.sh** → will only build all artifacts and run the tests. This script is also a subscript of the startup_local.sh script
- **clean_all_docker_images.sh**→ removes all docker images
- several minor convenience scripts

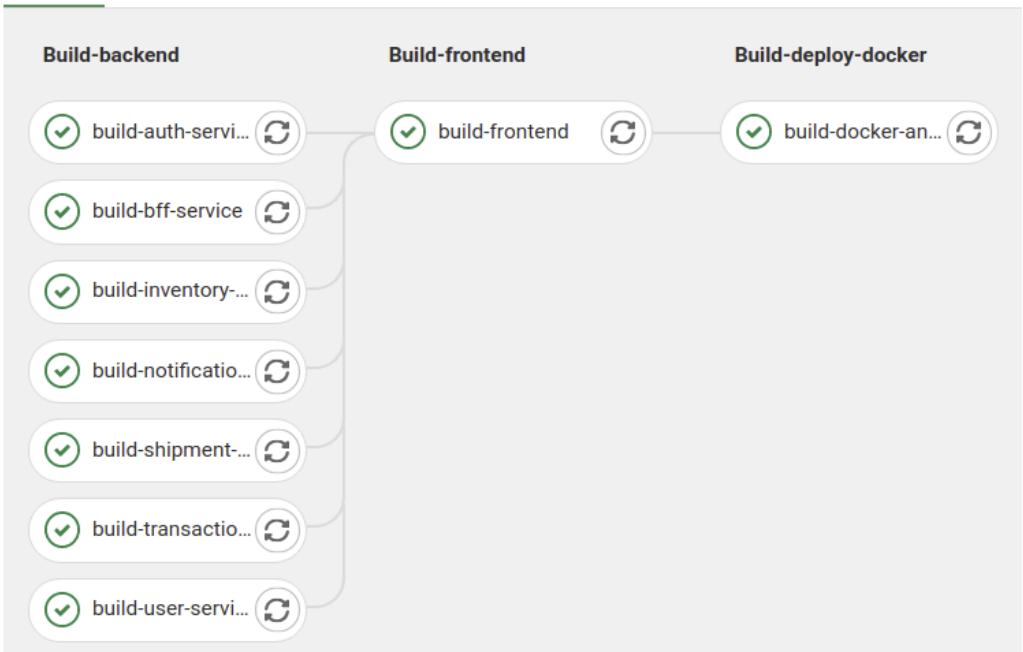
The purpose of these scripts is to easily startup the full environment or individual services at will. If individual services should be started, we simply leverage docker-compose and list the individual services:

docker-compose --env-file=./config/env.dev up kafka1

This allows for rapid testing and development at the local machine.

4.1. Continuous Integration

To enhance integration into the full code base, we also made use of the gitlab ci. For that purpose we configured the .gitlab-ci.yml to provide us with three distinct stages (similar to the local shell scripts):



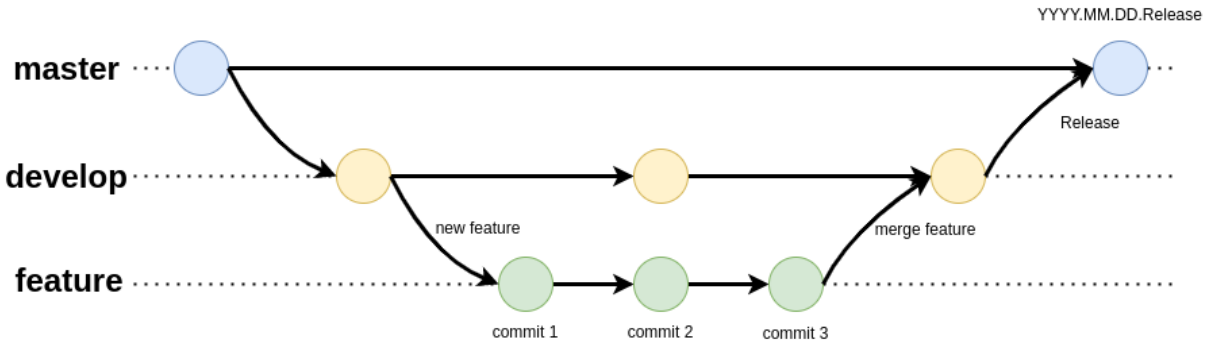
In these three distinct steps we perform a full backend build, a frontend build and a build step that combines all artefacts, creates docker images and packages them as a tar file. Every time the backend build stage is performed, each microservice is built individually, each unit test is performed and every integration test is executed. A failing unit test or a failing integration test causes the whole pipeline to fail. As soon as the pipeline fails, the developer whose commit caused the pipeline to fail, will be notified via email. A failing build cannot be merged into develop. More on the branching strategy here.

For the microservice integration tests to work in an isolated environment, we have added in-memory databases to every microservice and mocked away external dependencies, such as the Kafka event store, with a mock service. This is made possible by providing different configuration **profiles**, namely *dev* and *default*. Each profile defines which database to use and if we want to use our event store or the mocked version of it. The *default* profile is used if no other profile is provided. That also means, the production profile is *default*. For testing and development purposes, the *dev* profile is activated. Allowing developers to implement their functionality without depending on external services, such as kafka or the PostgreSQL database.

The aforementioned deployment.tar file can be used to install the complete environment on a testing environment. In this environment, a tester will perform a final acceptance test and will challenge the current feature with the acceptance criteria defined in the assignment sheet.

4.2. Branching Strategy

The **master branch** is our release branch. Every merge to the master branch represents an official release and is documented and distinguished with the title YYYY.MM.DD.Release. A release is issued by creating a merge request from develop to master. Each pull request is verified and tested by a human tester. A merge request can be rejected and declined.



The **develop branch** is our working/default branch and always contains a working and releasable version of our product. This branch is initially branched from master and just like master, will never be closed or deleted. We will add features to this branch until we have reached a milestone. After a milestone has been reached, we will create a merge request for release. Every merge to develop will trigger the full gitlab ci pipeline mentioned here. This will ensure that we always have a clean version on our working branch.

The **feature branch** is branched from develop. A software developer can work on a feature in a feature branch until the feature is complete. As soon as the feature is complete, the feature branch will be merged back to develop and deleted. Before the merge can happen, we require a code review and approval from a team member and successful execution of the gitlab ci build pipeline. For every feature branch we also create an issue in gitlab. This helps us track our progress and complete our product just like a checklist. Together with the issue number, we name our feature branches in this pattern: *feature/<developer initials>-<description>#<issue number>*

That means that we implement a feature in designated feature branches. When we consider the feature to be complete, we rebase against develop and resolve all merge conflicts. Thereafter, we create a pull request to develop. This will trigger a gitlab ci build. If the build pipeline is successful and the new code is approved, we merge the branch back into develop. As soon as we consider a significant milestone to be reached, we create a release pull request from develop to master. We tag this release and merge it to master as soon as the pipeline has succeeded one more time.

5. Team Contribution and Continuous development method

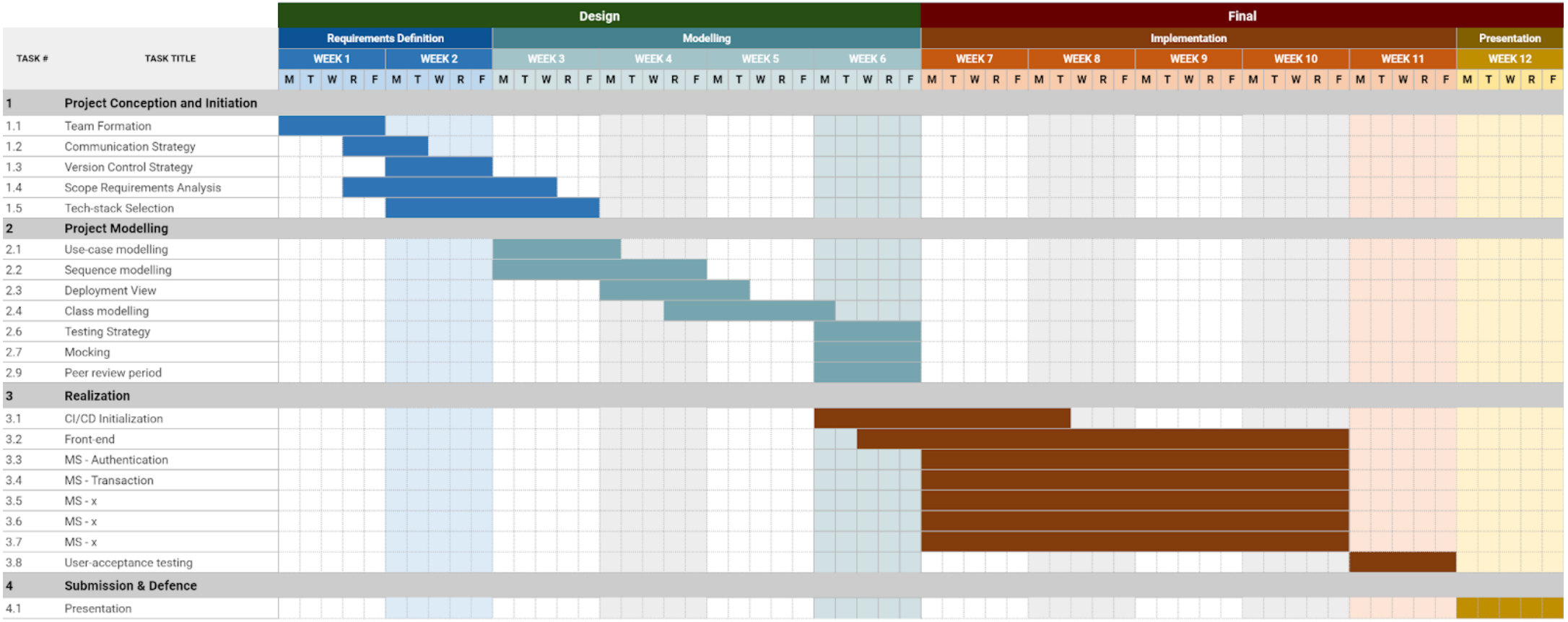
5.1. Project Tasks and Schedule

For the searchable distribution of project tasks our team has made use of the Gitlab issues provided by the University of Vienna. For each task a member has created an issue and then used the number assigned by the tracker, for naming the branch of interest where the task was implemented. The team has used

The following Gant chart depicts the phases of the project development as well as the general breakdown of tasks over the course of the semester.

GANTT CHART

Course: Advanced Sofware Engineering Team: 5 Semester: SS 2021



5.2. Distribution of Work and Efforts

INSERT WORK PROTOCOL HERE!!!

Team meets on a regular basis (using Jitsi meet) with respect to their time capabilities. During team meetings, each team member reports on the work done in the previous week. Further, for every meeting a responsibility is assigned to a team member to record the remarks and TODOs in form of markdown notes which he then posts to the GitLab Wiki repository. During the session, all team-members actively participate in reviewing the presented material (code, models, factual information etc.). At the end of the session, noted tasks are broken down into logical parts and assigned to team members with the deadline being the next meeting.To support this, issues are assigned by the note-taking teammember to track the progress and current state of the project.

Contribution of *Zemanec Hynek*:

Scope requirements responsibility:	• **SR3** • **SR8**
Code Ownership	• **Inventory Product MS** • **Shop Front-end**

Contribution of *Puri Himal*:

Scope requirements responsibility:	• **SR6** • **SR7**
Code Ownership	• **User MS**

Contribution of *Preisinger Johannes*:

Scope requirements responsibility:	• **SR4** • **SR5**
Code Ownership	• **Payment Transaction MS** • **Authentication MS** • **BFF MS**

Contribution of *Lascsak Christian*:

Scope requirements responsibility:	• **SR9** • **SR11**

Code Ownership	• **Notification MS** • **Price Suggestion Component**

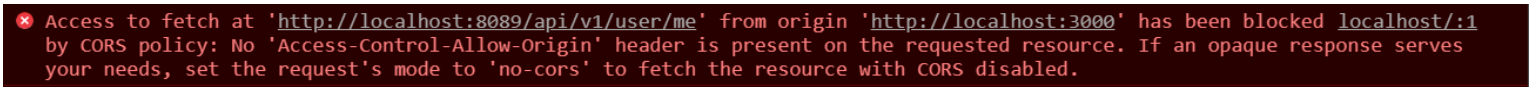
Contribution of *Kapeller Angelika*:

Scope requirements responsibility:	• **SR10** • **SR12**
Code Ownership	• **Shipment MS**

6. How-to documentation

Executing application in the browser

For security reasons, modern browsers disallow fetching of the resources that do not originate from the same URL, also known as cross origin resource sharing (CORS). This policy applies for services running on the local machine as well, i.e. fetching API resources from `localhost:8089` while serving the front-end from the `localhost:3000` tends to result in the following message:



In order to circumvent this issue and run the system on the local machine, it is necessary to allow cross origin resource sharing in the browser of choice. For Chromium related browsers (tested on Chrome on Windows 10), it is necessary to first **close all instances** of Chrome and then start the Chrome browser using the command below:

```
chrome.exe --user-data-dir="C:/Chrome dev session"
--disable-web-security
```

After that navigate to http://localhost:8089/ which will automatically redirect you to the login page on http://localhost:4001/auth/login (see login credentials below). After successful login, depending on the user-role, the front-end becomes available on http://localhost:3000/.

Please note that this solution is required only for the local settings in which all Microservices are running on the local machine. The problem with CORS would be omitted in a scenario where the system is deployed on a server and accessible from the internet.

How to execute the remote pipeline:

Start CI/CD Pipeline

If you start this pipeline, each application will be built and automatically tested. The tests include unit tests and integration tests by default. All of them are triggered during the build phase. The docker images will be created and the images will be combined into a deployment.tar.gz file. To explore the app locally, look at the steps below.

Step by step guide to explore the endpoints we support

- Go to ASE_Team_0501/
- Execute: **startup_local.sh**
- Wait until the **electro_ui** starts. It will be the last container to start
- Visit: http://localhost:8089/swagger-ui.html
- You will be redirected to the login page. We prepared a user for you
  - Username: evangelos.ntentos@univie.ac.at
  - Password: admin
- You will be redirected to the swagger ui and can now experiment with the supported endpoints.

How to exercise the endpoints of the transaction service?

All the supported getters actually work with the uuids of either seller, buyer or user.

You may want to start by calling GET `/api/v1/transactions` to get an overview of all currently existing transactions in the system. We migrate the database with one initial transaction at the start. You needn't fill in any data for this endpoint to work.

For more interesting or longer results, execute POST `/api/v1/transactions`

We prepared following payload to exercise this endpoint:

ASE_Team_0501/implementation/e2e-testing/transactions/post-transaction.json

Afterwards you can call either of these endpoints to filter for the transactions:

GET `/api/v1/users/{userId}/transactions`

5a48c43f-4c24-4ee5-bf75-430f83153349

GET `/api/v1/sellers/{sellerId}/transactions`

5a48c43f-4c24-4ee5-bf75-430f83153349

GET `/api/v1/buyers/{buyerId}/transactions`

6d8ea0ea-1162-4739-87b4-6729ef7f1a20

These endpoints will filter the results according to the role encoded in the path. The endpoint user will return all the transactions the user is involved in.

Last but not least, you can calculate the profits for following uuid:

GET/api/v1/sellers/{sellerId}/profits

5a48c43f-4c24-4ee5-bf75-430f83153349

Of course, you can shuffle the uuids for the request and see how the results are changing.

How to exercise the endpoints of the notification service?

GET/api/v1/productmarking/user/{userId}

Gets the marked products for the specified user id.

POST /api/v1/productmarking/

Adds a new Product Mark. A product mark contains

- userid
- productid
- emailAddress
- price

DELETE/api/v1/productmarking/

How to exercise the endpoints of the shipment service?

GET/api/v1/shipments

To get all shipments stored in the database.

POST/api/v1/shipment/{uuid}/status/{status}

To trigger a shipment status update.

A possible UUID is: **008865a4-8348-11eb-8dcd-0242ac130003**

The shipment update status can be:

- **OrderConfirmation**
- **OutForDelivery**
- **Delivered**.

The purpose of this endpoint is to mock the status update of a shipment.

How to exercise the endpoints of the inventory service?

Inventory

GET /api/v1/vendor/{vendorId}/inventory

To get the inventory of a vendor.

Possible vendorId: `d7a8dc0b-c843-4882-8e9d-43dde1dddcd2`

PUT /api/v1/vendor/{vendorId}/inventory

To update the inventory of a vendor. It is possible to increase/decrease stock.

Possible vendorId: `d7a8dc0b-c843-4882-8e9d-43dde1dddcd2`

Product

PUT /api/v1/products/{uuid}/rating

To update the rating of a product.

Possible uuid: `321f737d-f8dd-4c9d-aca8-b8c6164a1dd1`

GET /api/v1/products/{productId}

To get the product information of a product.

Possible productId: `dfc6d5d8-3d15-4d8a-a29a-01f5894e122e`

PUT /api/v1/products/{productId}

To update the product information of a product.

Possible productId: `dfc6d5d8-3d15-4d8a-a29a-01f5894e122e`

PUT /api/v1/products/{productId}/price

To update the price of a product.

Possible productId: `dfc6d5d8-3d15-4d8a-a29a-01f5894e122e`

GET/api/v1/products

To get all products stored in the database.

POST/api/v1/products

To add a new product to the database.

How to exercise the endpoints of the user service?

The user's service's endpoints work with the uuid of user and product. There is no payload in any of the endpoints. The requirement information can be provided via the path variable in the url.

GET/api/v1/users

This URL will return all the users present in the user ms. Along with the cart and products in the cart. At the start all the carts are empty. I added the 14 users using flyway script, which will display all the users.

GET/api/v1/users/{userId}

This URL will return a single user with its cart and products in it. You should call with the user uuid which is present in the user service.

d7a8dc0b-c843-4882-8e9d-43dde1dddcd2

GET/api/v1/users/{userId}/cart

This url returns a cart of the user without user details.

POST/api/v1/api/v1/users/{userId}/cart/product/{productId}

url This will save the product in the user's cart in user service. user uuid and product uuid should be passed via the url. If the product is already present, the count will be increased in the cart.

product id: 321f737d-f8dd-4c9d-aca8-b8c6164a1dd1

user id: d7a8dc0b-c843-4882-8e9d-43dde1dddcd2

`DELETE /api/v1/api/v1/users/{userId}/cart/product/{productId}`

This is used to delete the already present cart of the user. product id and user uuid is required for this purpose.

How to locally deploy the application into the deployment folder:

Go to ASE_Team_0501/

- Execute: build.sh
- Execute: **deploy.sh**

This will store the complete deployment in the the deployment.tar.gz

And if you would like to start the environment locally, go to the deployment folder:

- Execute: `docker load < deployment.tar.gz`
- Execute: `docker-compose --env-file=./config/env.dev up`

The docker-compose file for the deployment is also located in the deployment folder.

How to start the integration tests:

The integration tests are executed by default at every build. They are part of the individual microservices. The correct profile for integration testing is chosen automatically.