

Advanced Software Engineering

DESIGN REPORT

Team number:	5
---------------------	---

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

Team member 2	
Name:	Puri Himal
Student ID:	11738090
E-mail address:	a11738090@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:	Lascsak Christian
Student ID:	01363742
E-mail address:	a01363742@unet.univie.ac.at

Team member 5	
Name:	Kapeller Angelika
Student ID:	01268478
E-mail address:	a01268478@unet.univie.ac.at

Design Draft	5
Design Approach and Overview	5
Assumptions	5
Project goal	5
Target Group - Customers	5
Target Group - Vendors	5
Location	5
Limitations and future goals	5
Stakeholders	6
Design Decisions	7
Notification Service vs No Notification Service	7
User Microservice vs No User Microservice	7
Synchronous vs Asynchronous Communication for Querying Data	8
BFF does not communicate with message Broker	8
Synchronous vs Asynchronous Communication	9
Iteration of sequence diagram over the design phase.	9
Design Overview	12
Development Stack and Technology Stack	14
Development Stack	14
Technology Stack	14
System Requirements	16
4+1 Views Model	19
Scenarios / Use Case View	19
Use Case Diagram	19
Use Case Descriptions	20
Logical View	27
Authentication	27
Payment Transaction	28
3.2.3 Shipment	29

3.2.3. Product Inventory	30
3.2.5 Notification	31
3.2.6 User	32
Process View	33
Development View	34
Physical View	35
Team Contribution and Continuous development method	36
Project Tasks and Schedule	36
Continuous Integration, Delivery and Deployment Plan	37
Distribution of Work and Efforts	38
Contribution of Zemanec Hynek:	38
Contribution of Puri Himal:	39
Contribution of Preisinger Johannes:	39
Contribution of Lascsak Christian:	39
Contribution of Kapeller Angelika:	40
Team effort time estimates	40
How-to / mock-up documentation	43
Payment Transaction Service	44
Notification Service	45
Product Inventory Microservice	46
All Products	46
Product	46
Vendor's inventory	47
Create Product	47
Update Product	47
Update Inventory	47
product.update	47
Shipment Microservice	48
User MS	49

Design Draft

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 or other languages.

- 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

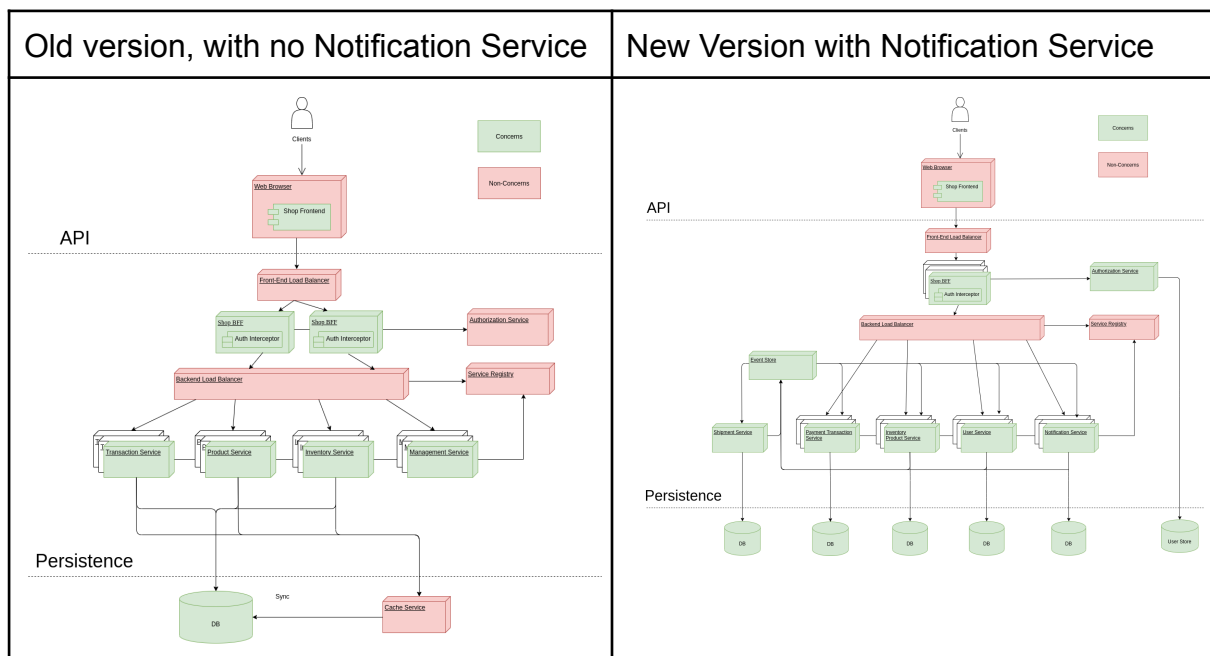
- Webshop company
- Breweries
- Customers
- Transport suppliers
- Competitors
- Client competitors
- Developer team
- Operations
- DevOps
- Test team
- Marketing
- Legal department
- Customer Service

1.1.2. Design Decisions

Notification Service vs No Notification Service

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.



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

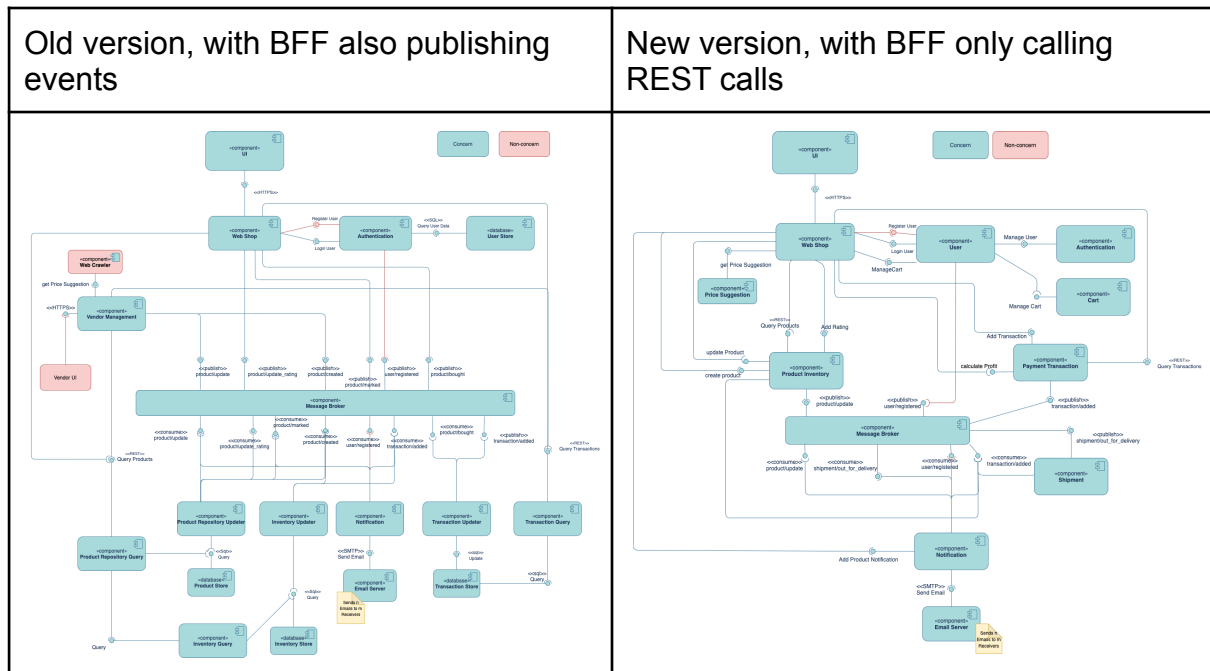
Firstly we designed the querying for products in an asynchronous way, via the message broker. This means that a message would be published, requesting the products. The product service would consume this event and publish a new one with the queried products (see Process View).

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.

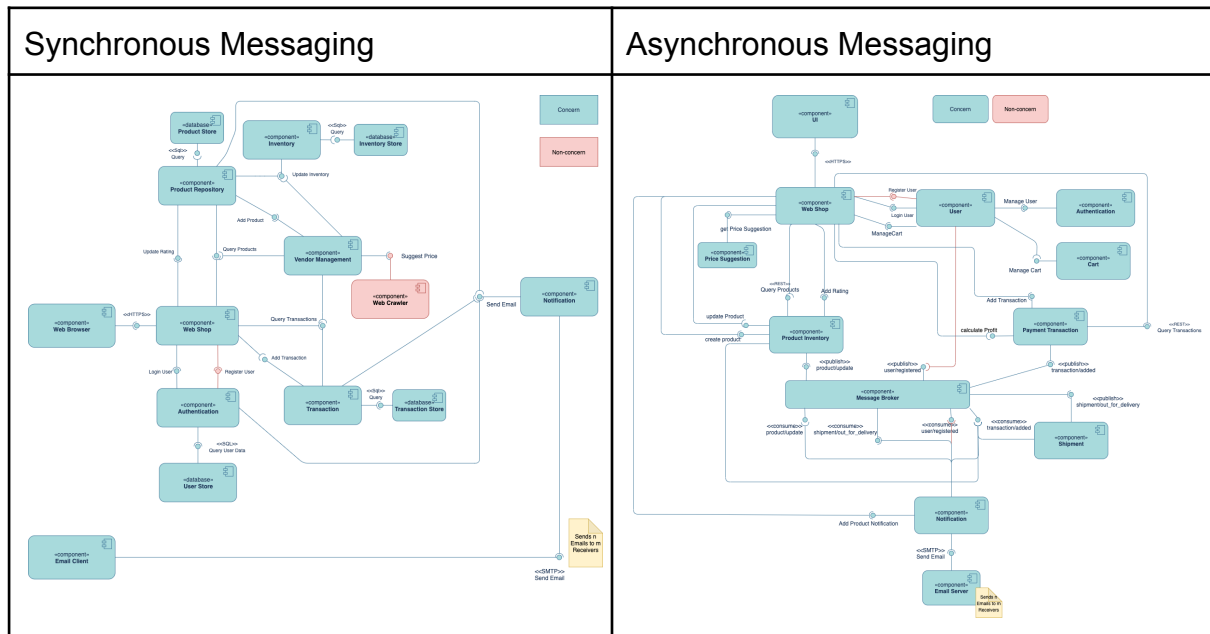


Synchronous vs Asynchronous Communication

In the first iteration, we designed the system to communicate completely synchronous 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 be reflected in the Component Diagram:



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 of 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 kafka message broker that will have asynchronous communication with the other MS.

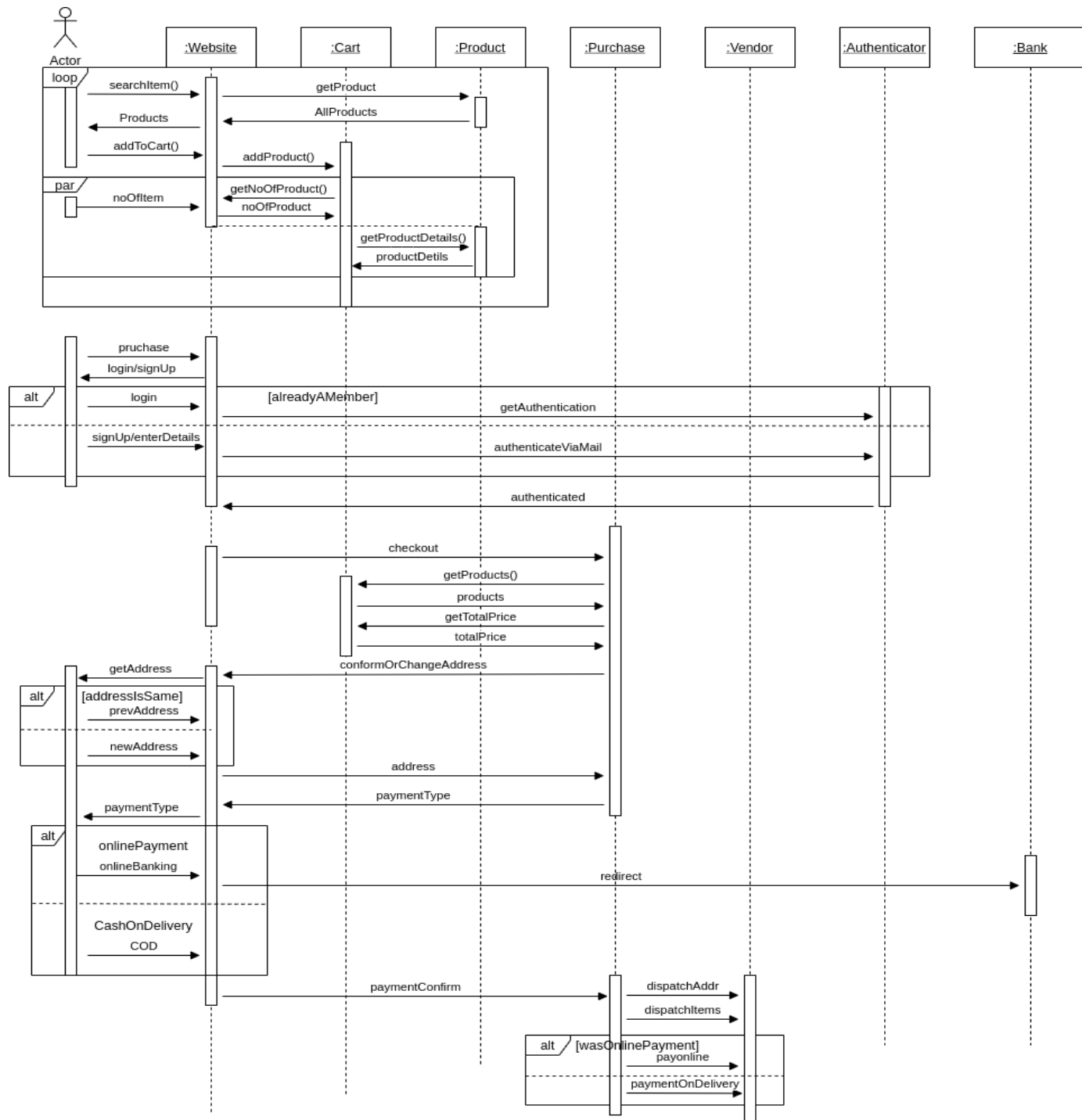


fig1: First iteration of the sequence diagram design

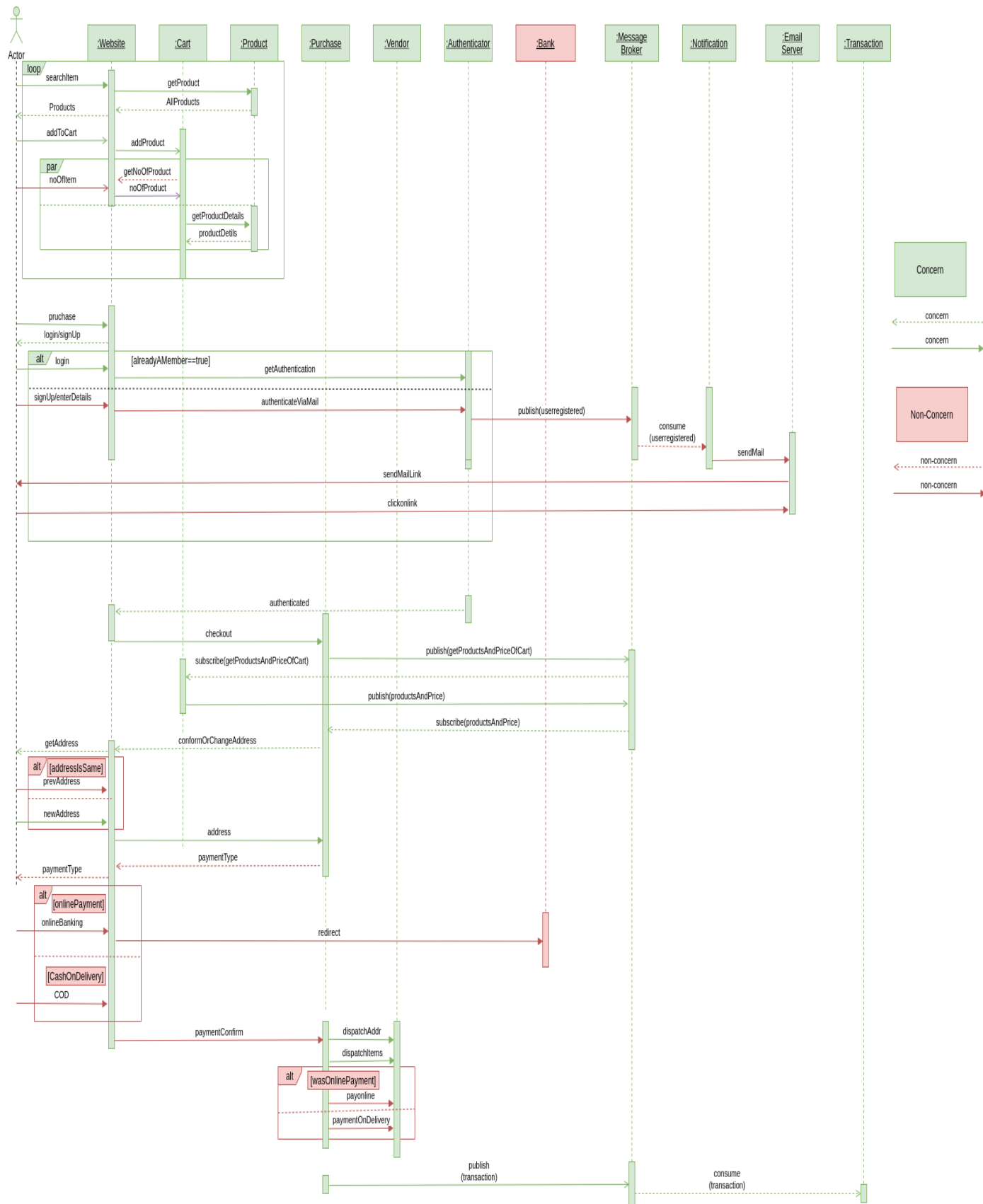


fig2: Final sequence diagram design.

1.1.3. Design Overview

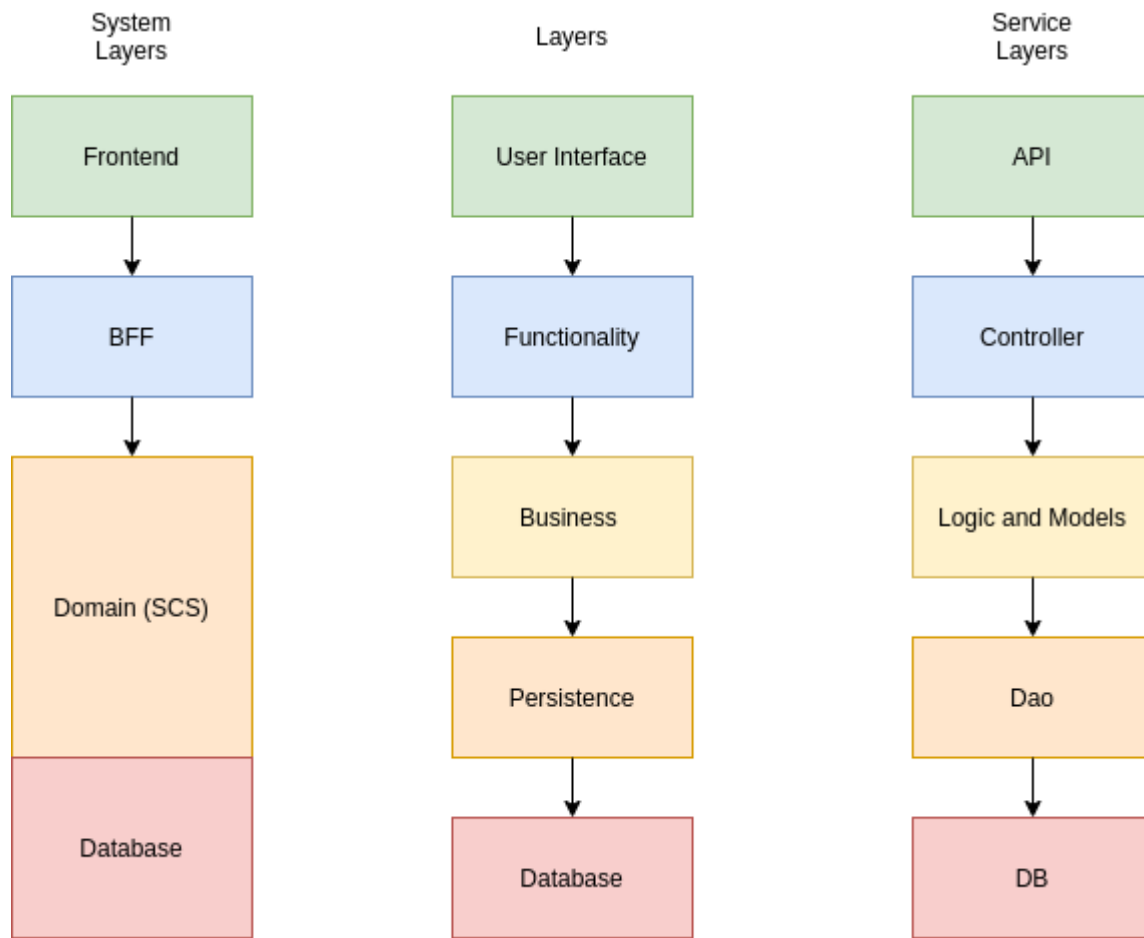
We decided to look at our problem domain and separate it into smaller distinct domains that can function and work independently from each other. If communication between domains is required, they will do so, by issuing an event to a central Event Store. Every domain interested in this event only needs to know the Event Store and no other domain. That will reduce coupling between these domains and guarantee that domains can be added or removed independently.

Each domain can be delivered as a self contained system. It can operate by itself, can be started by itself and therefore be easily reused if the need should arise. For that reason, the service must be delivered with it's own datasource.

An additional advantage of this approach is, that each self contained system can be scaled independently from each other. A transaction service might scale very differently from a product service and could therefore be deployed and scaled individually. For that reason we identified several core domains in our system, which we will discuss in more detail in future chapters, and created a self contained system for each of them. For communication we will depend on an Event Store that will make use of a Publish-Subscribe pattern, but on a microservice level.

To guarantee that each client request is authenticated, all client requests will be channeled through a Backend For Frontend service which is uniquely designed to service one specific Frontend Client and offer it a unified API. That will "hide" for the frontend application the fact that we have multiple microservice that implement the same API. This will further allow us to intercept each client request with an authentication interceptor and delegate the authentication responsibility to an authentication server. If there would be a need for another frontend, say an inventory system for the fulfillment centers, another BFF could be developed for this frontend, offering a different set of interactions and utilizing a different set of our self contained systems.

Such as the system overall, also each of the individual self contained systems will be separated into multiple layers of responsibility. The layering will divide as follows:



This layering will help the individual teams to focus on the implementation and not on how to structure the code. It will serve as a guide for the System Design and Package structure of the individual Microservices.

Because no knowledge of each other is required, each individual Self Contained System can be developed, tested, deployed and maintained by different teams, without having to rely on each other.

1.2. Development Stack and Technology Stack

1.2.1. 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)

1.2.2. 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. System Requirements

ID	Description	Implication on implementation	Verification
Availability	Our shop must offer interested customers a certain amount of up-time. Therefore the application must be robust. Because at the beginning, we are only a regional e-Shop, we aim to reach two nines (99%) of availability.	Therefore, we must build our system to be scalable. We will use docker to package our 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.	Monitoring tool, that will track the amount of transactions being 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 will 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	During development, we will use a security checklist, containing the most important dos and don'ts. Each tester will be provided with a

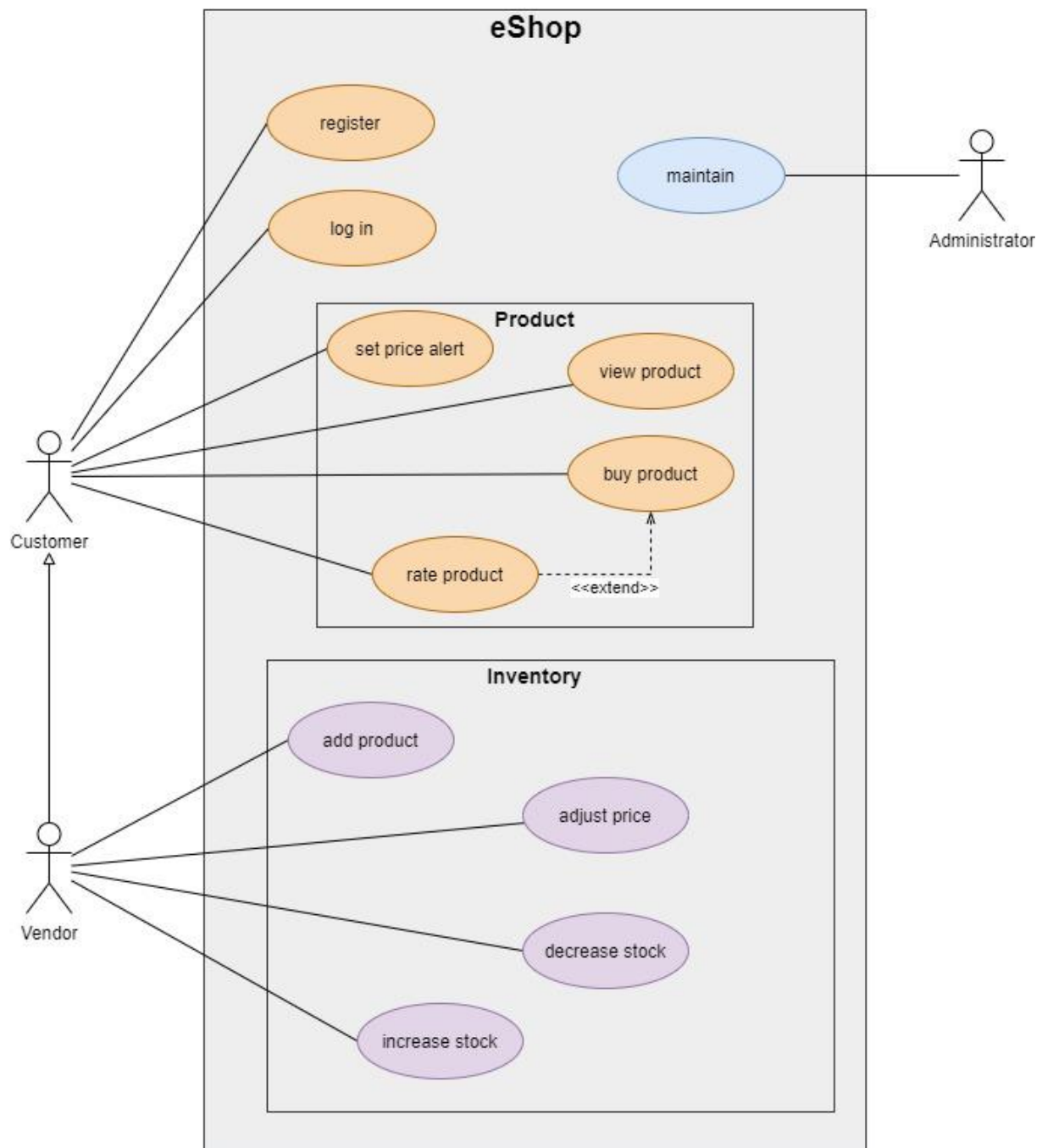
		supported, we use bcrypt encryption. It supports 64 characters and it will make sure that we never store the passwords as is, but just an encrypted representation of it. All requests being made to our environment, need to pass through the BFF. The BFFs will be the main authority to check user requests for a valid authenticated token. We will use a server side authentication mechanism, offered by Spring Boot security. and a login page rendered server side.	similar security checklist and perform regular checks. We will also try to make use of automatic security scanning tools, such as SonarQube for Security Flaws in our Codebase. Pre-Release we will 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 will make sure that our code is modular and tested.	We will measure our unit test coverage and run the code through a linter and SonarQube, to verify that coding standards are applied.
Reliability	Our services need to be reliable by providing a 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	We will run performance and stress tests on our system. We will also implement a monitoring tool, in which we can see

		load is evenly distributed and therefore increases the reliability of the system.	the load compared to response times and resources of the system.
Shipment	A user receives an email when their shipment status changes	We will have a notification service that listens to changes of a shipment update and will send notification emails to the users	We will set up a test mailbox that receives shipment update Emails.
Functional Requirements	As described by the use cases	See use cases	See use cases
System Configuration Management	The system should be easy configurable, and run on multiple machines	We use Docker in our microservices, which allows them to be independent of the hosts system. It also serves as a central point for configuring the different containers.	System can be run on different hosts 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.	

3. 4+1 Views Model

3.1. Scenarios / Use Case View

3.1.1. Use Case Diagram



3.1.2. Use Case Descriptions

Use case:	Register
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:	<ol style="list-style-type: none"> 1. The Customer selects "create account". 2. The Customer is redirected to the registration page. 3. The Customer fills in their email address (= 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:	<ul style="list-style-type: none"> • Password restrictions need to be defined. • Password should be entered twice. • The verification link shall expire after a set amount of time. • Without verification, a user might not buy products.

Use case:	Log In
Use case ID:	UC 02
Actor(s):	Customer
Brief description:	The Customer logs in to the eShop.
Pre-conditions:	<ol style="list-style-type: none"> 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:	<ol style="list-style-type: none"> 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 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:	<ul style="list-style-type: none"> • If the user fails three login attempts, they have to wait for one minute for another login attempt. • Should we make a bot check?

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:	<ol style="list-style-type: none"> 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:	<ol style="list-style-type: none"> 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:	How to deal with deleted products with set alert?

Use case:	View product
Use case ID:	UC 04
Actor(s):	Customer

Brief description:	The Customer views a product in the eShop.
Pre-conditions:	<ul style="list-style-type: none"> • 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:	<ol style="list-style-type: none"> 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:	<ul style="list-style-type: none"> • 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:	What information is displayed on the product page?

Use case:	Buy product
Use case ID:	UC 05
Actor(s):	Customer
Brief description:	The Customer buys a product.
Pre-conditions:	<ol style="list-style-type: none"> 1. The Customer is logged in. 2. The Customer views a product.
Post-conditions:	The product is shipped.
Main success scenario:	<ol style="list-style-type: none"> 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 is prepared for shipment. 7. The Customer will get a notification that a product is ready for shipment. 8. The Customer will get a notification that a product is being shipped.
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:	Does the stock of the Vendor decrease automatically?

Use case:	Rate product
Use case ID:	UC 06
Actor(s):	Customer
Brief description:	The Customer rates a purchased product.
Pre-conditions:	<ol style="list-style-type: none"> 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:	<ol style="list-style-type: none"> 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:	Is it possible to edit or delete the rating?

Use case:	Add product
Use case ID:	UC 07
Actor(s):	Vendor
Brief description:	The Vendor adds a product to the inventory.
Pre-conditions:	<ul style="list-style-type: none"> • Vendor accounts need to be in the database. • The Vendor is logged in.
Post-conditions:	The product appears in the eShop.
Main success scenario:	<ol style="list-style-type: none"> 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:	<ul style="list-style-type: none"> • 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:	<ul style="list-style-type: none"> What information is required to add a new product (name, price, stock, ...)? How many products can be stored in the inventory?

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:	<ul style="list-style-type: none"> 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:	<ol style="list-style-type: none"> The Vendor clicks on "adjust price". The Vendor receives a suggested price based on the result of a web crawler. The Vendor accepts the suggested price or chooses to select a different price. The Vendor saves the updated product information. 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:	<p>Is the web crawler service mocked?</p> <p>If yes, how is it mocked? Randomized prices?</p>

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:	<ul style="list-style-type: none"> 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:	<ol style="list-style-type: none"> 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:	<ul style="list-style-type: none"> • Is the available amount of a product displayed for the Customer or just if/if not product is available? • Deadlock in the event of customer buying a product while the vendor is managing the inventory. • Concurrent purchase of n customers.

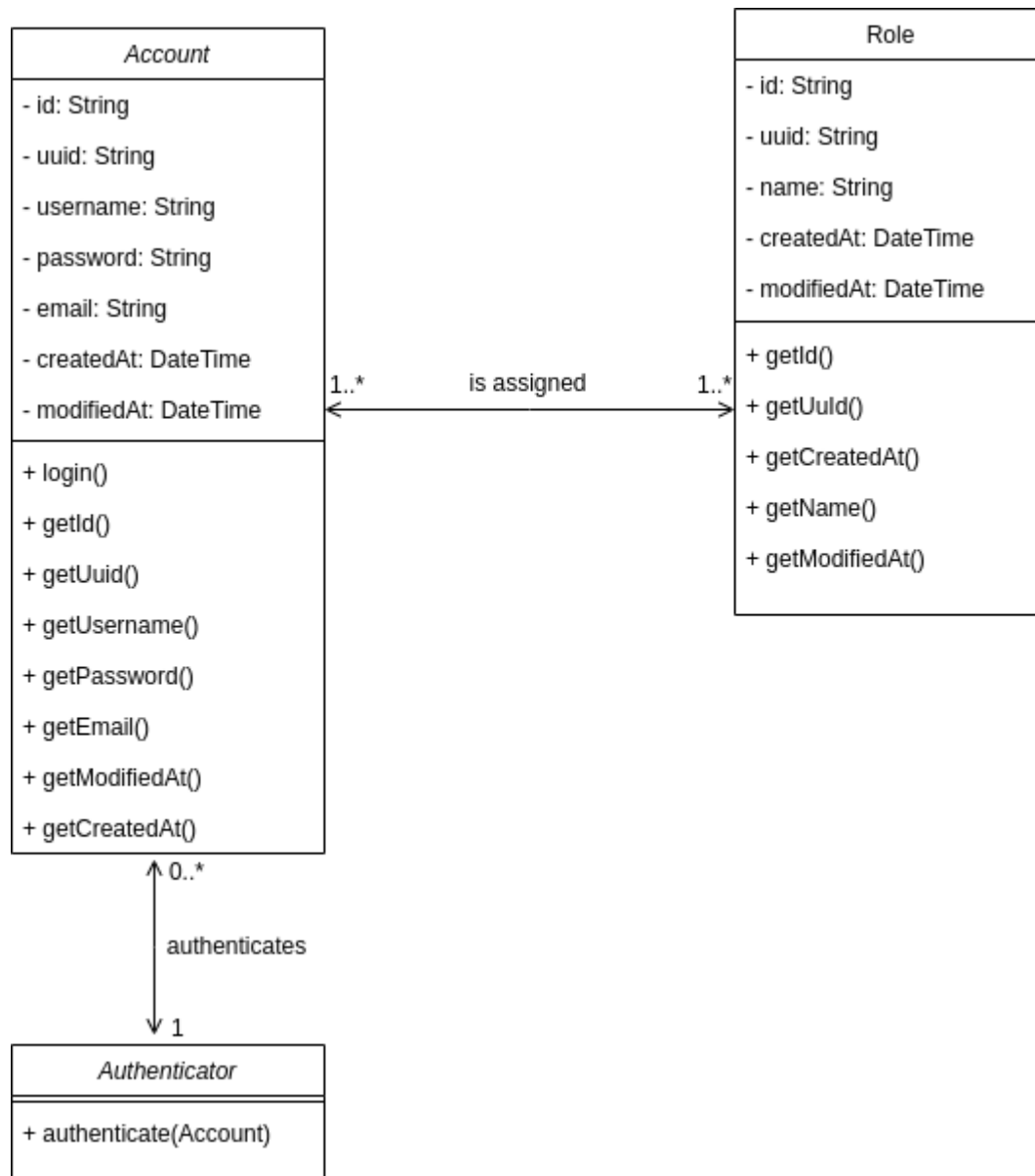
Use case:	Increase stock
Use case ID:	UC 10
Actor(s):	Vendor
Brief description:	The Vendor increases the availability of a created product in the inventory.
Pre-conditions:	The Vendor is logged in. The product stock is owned by the Vendor. The product stock is available (was created).
Post-conditions:	The product is displayed with the updated availability.
Main success scenario:	<ol style="list-style-type: none"> 1. The Vendor will call "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:	-
----------------	---

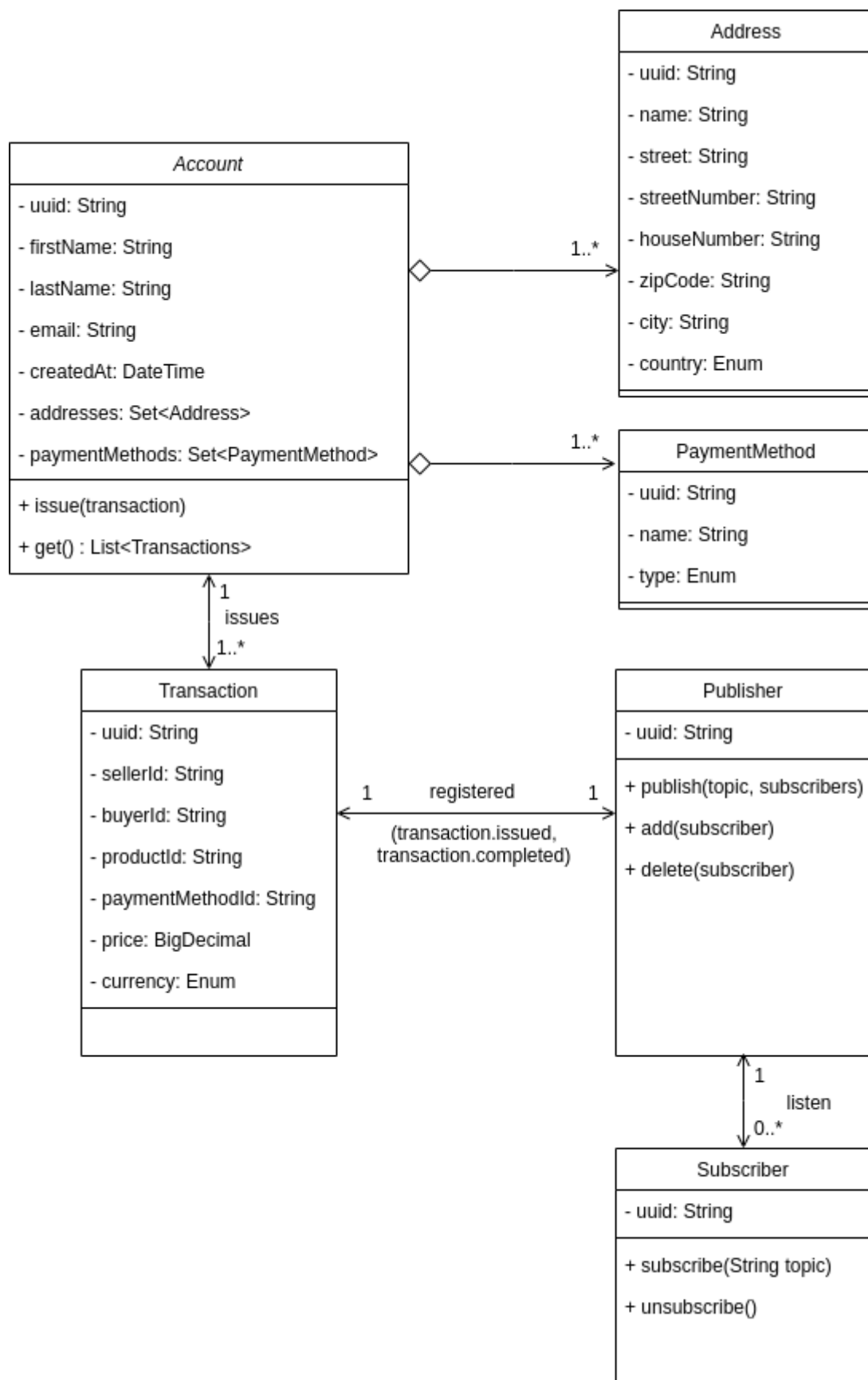
Use case:	Maintain
Use case ID:	UC 11
Actor(s):	Administrator
Brief description:	The Administrator maintains the system by updating and deploying a service to the eShop.
Pre-conditions:	<ol style="list-style-type: none"> 1. New code is available. 2. The new code was tested. 3. The Administrator is logged 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. This use case is necessary to maintain and refine the eShop.
Performance target:	-
Issues:	-

3.2. Logical View

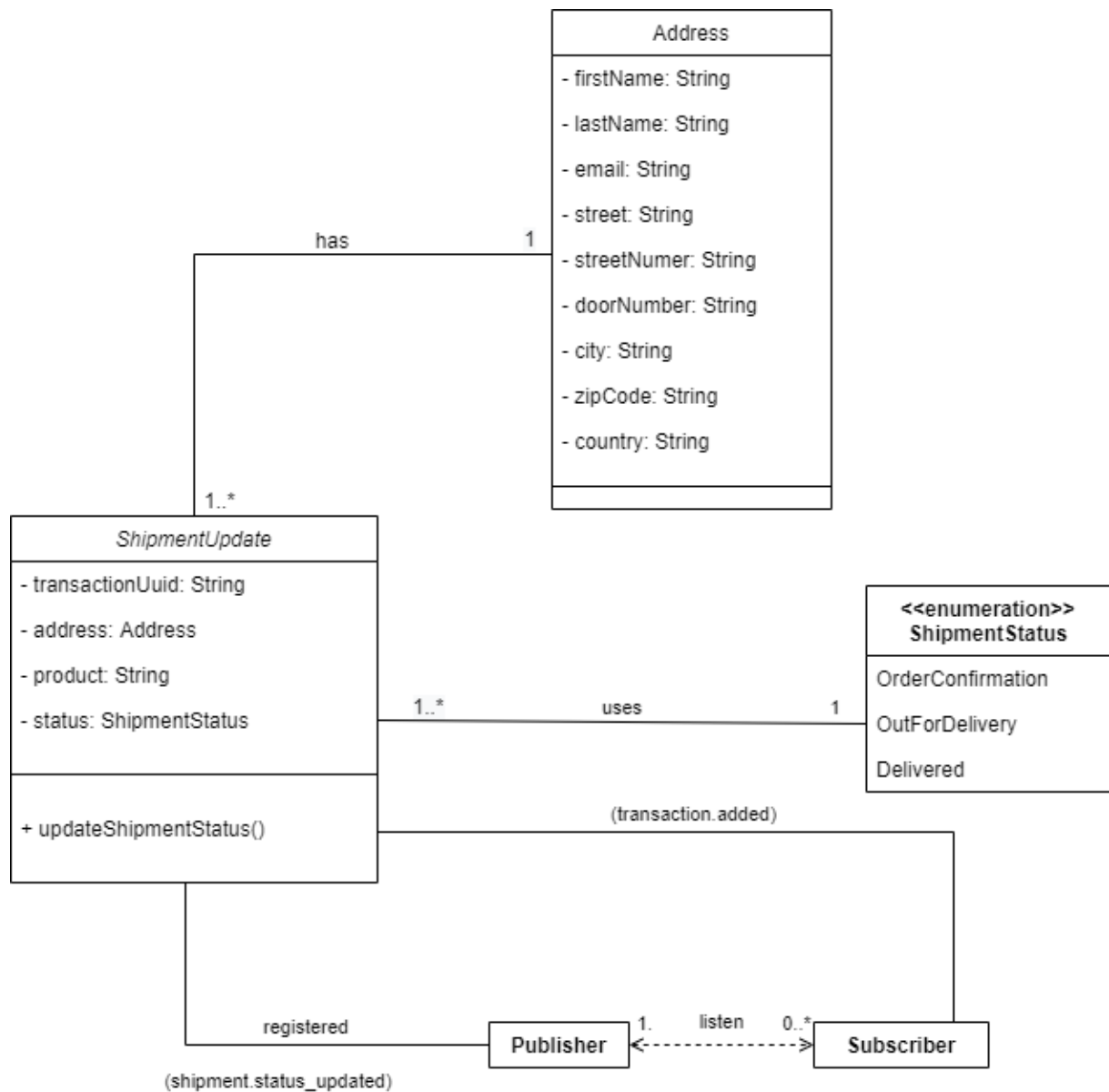
3.2.1. Authentication



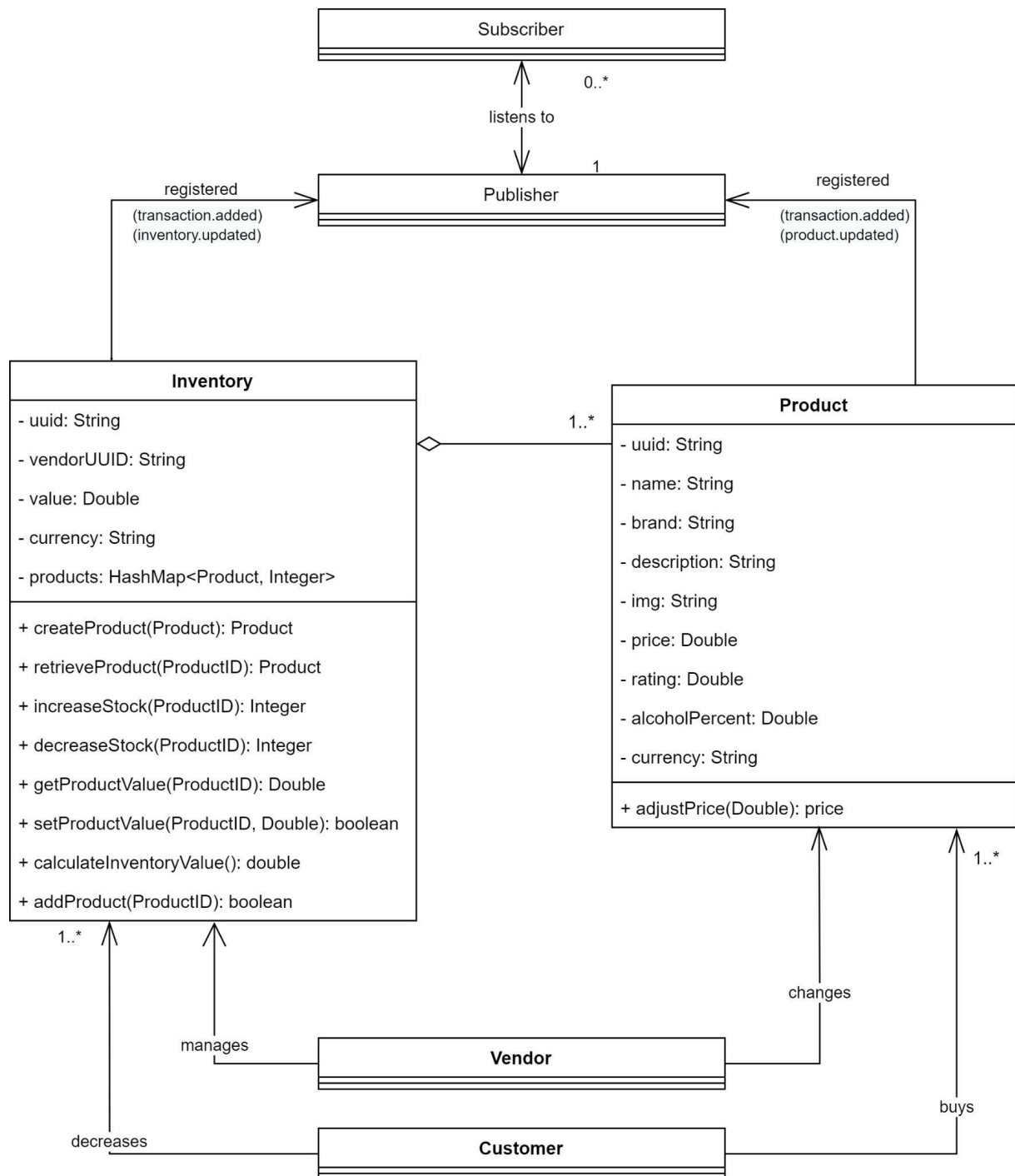
3.2.2. Payment Transaction



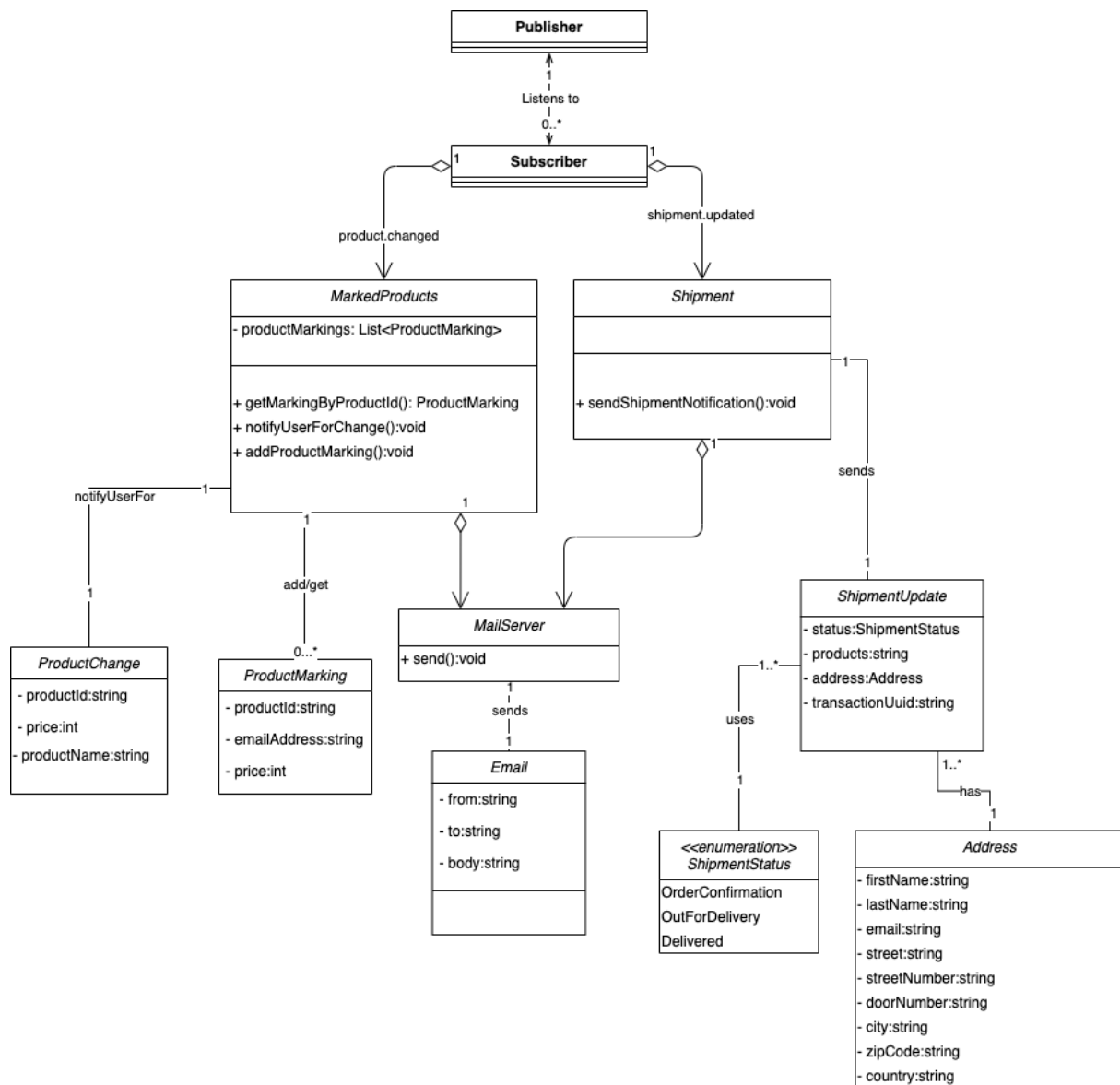
3.2.3 Shipment



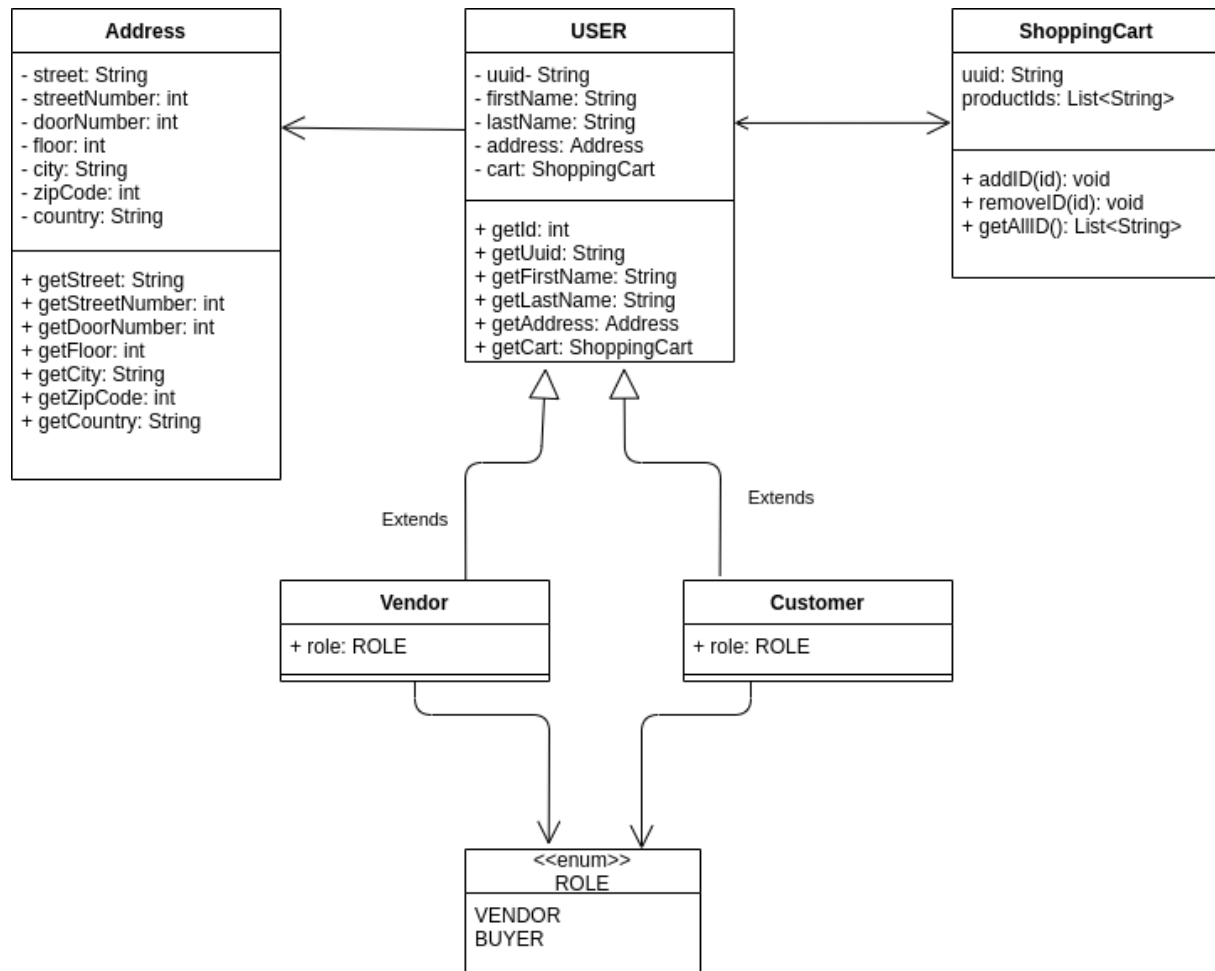
3.2.3. Product Inventory



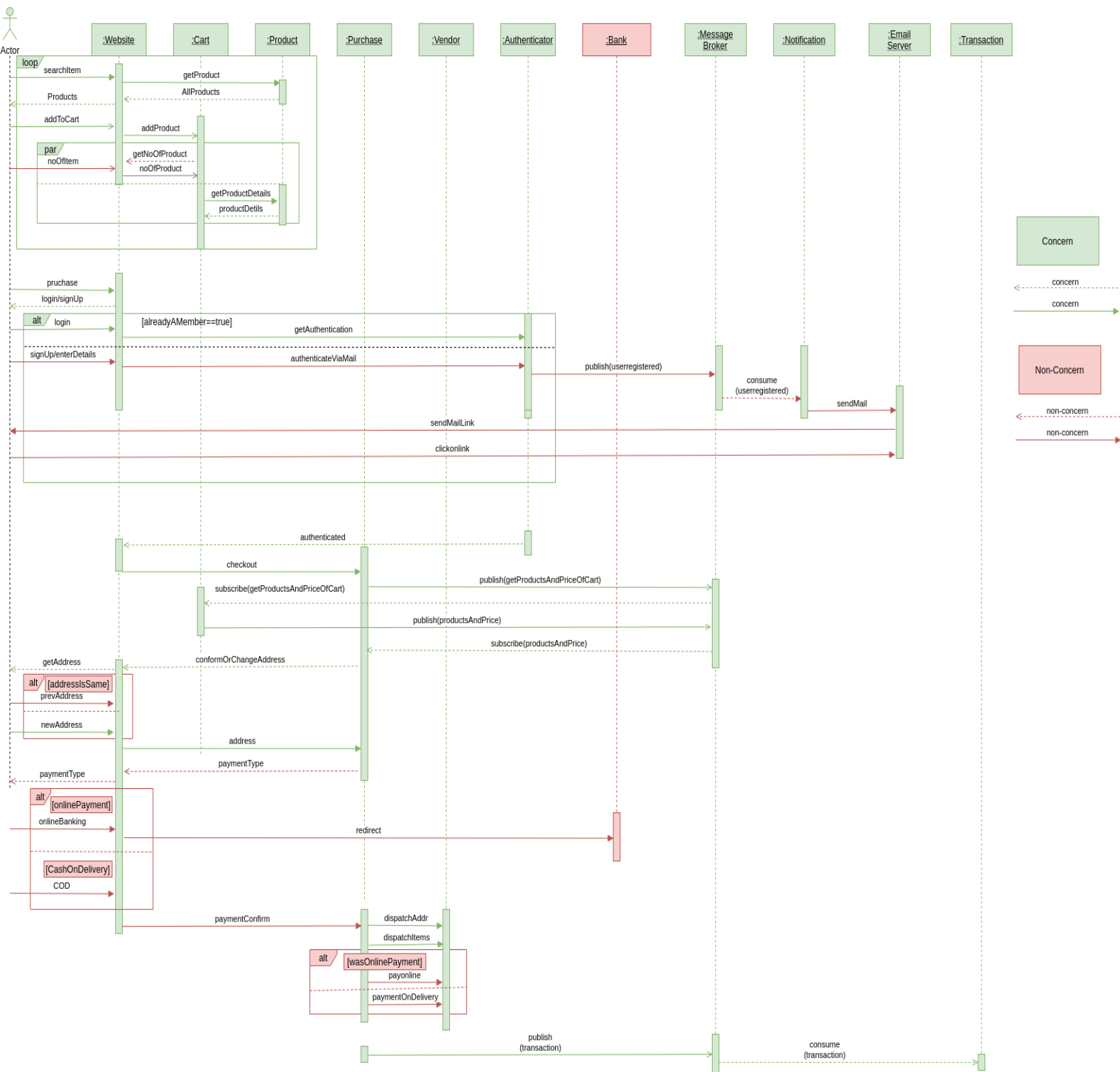
3.2.5 Notification



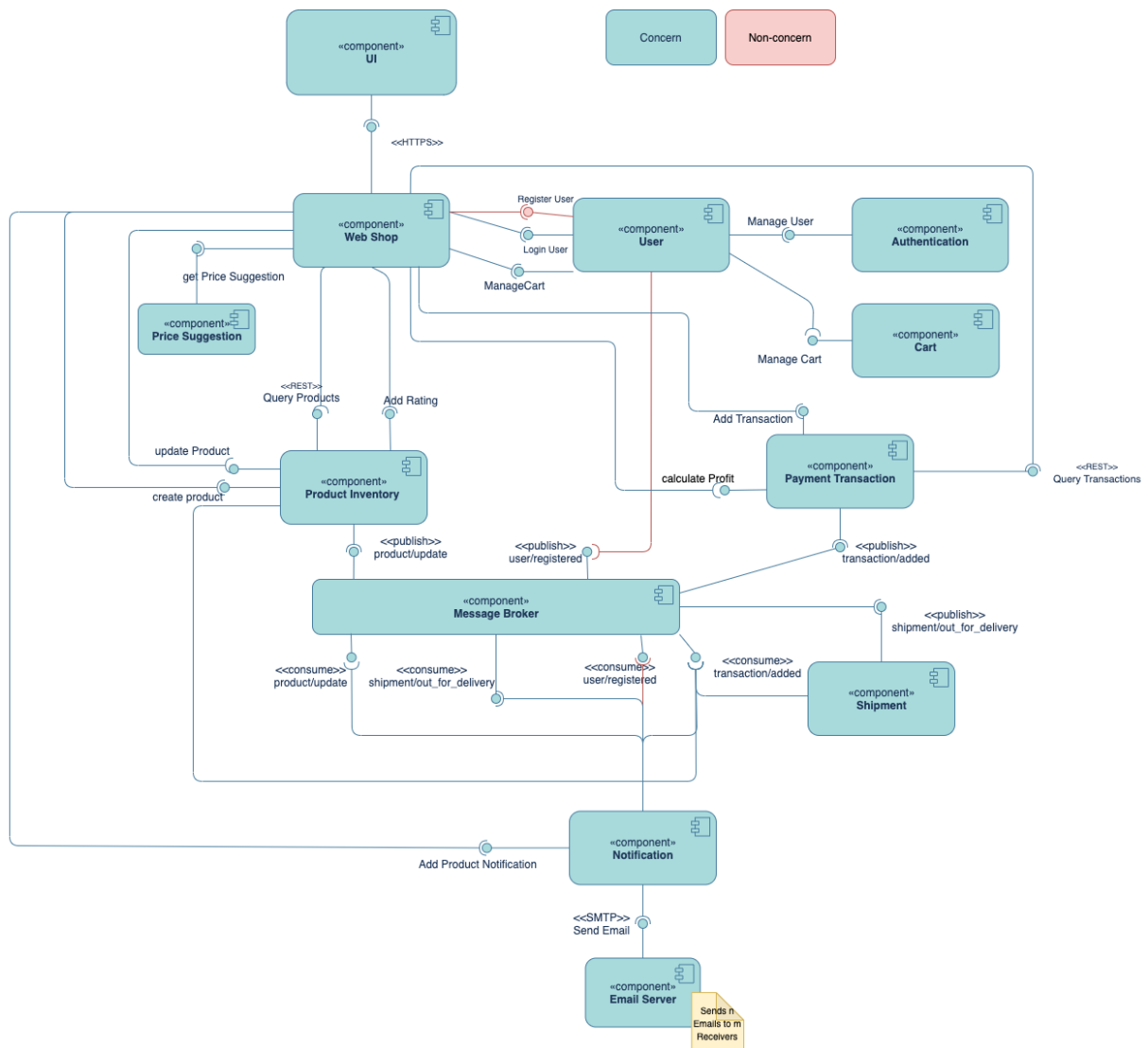
3.2.6 User



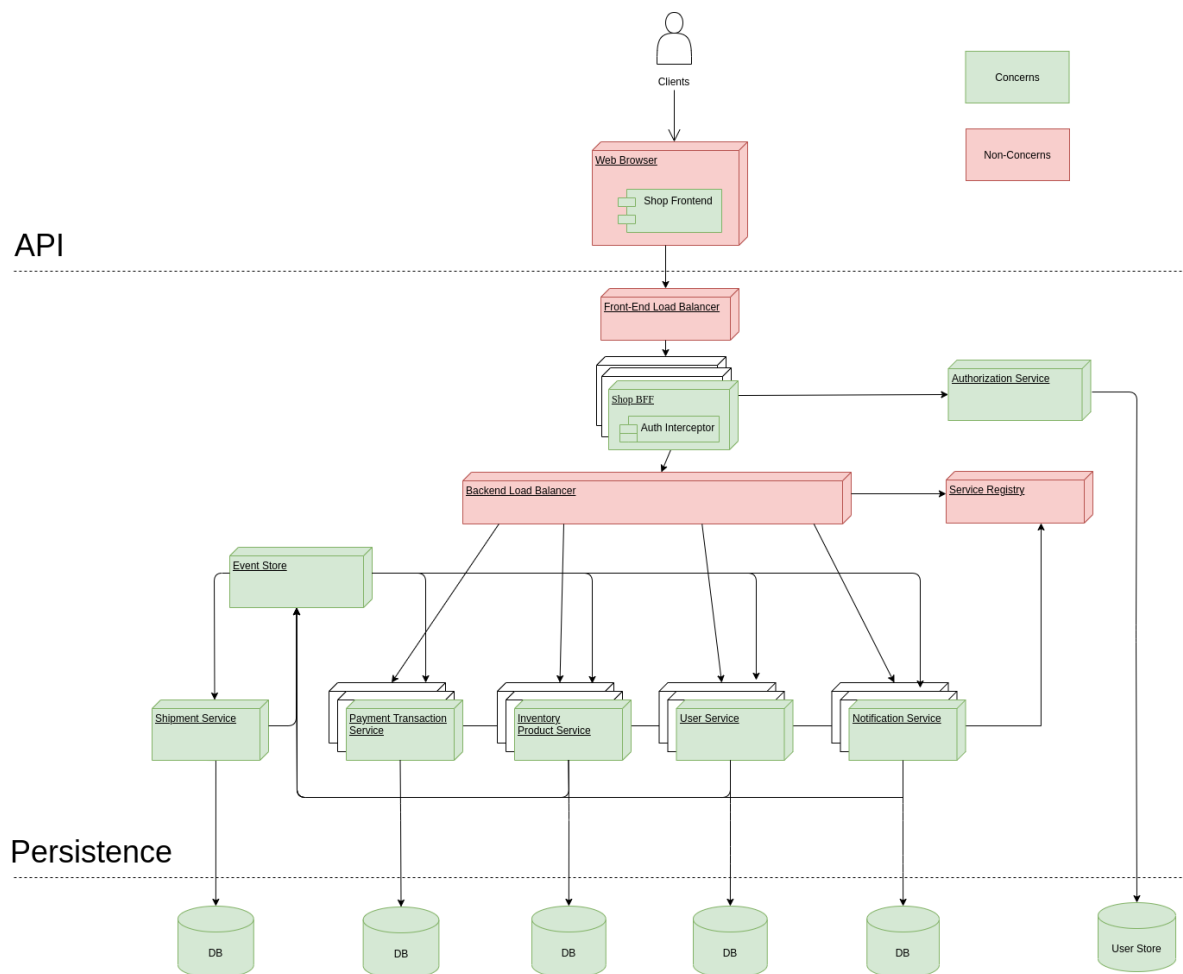
3.3. Process View



3.4. Development View



3.5. Physical View



4.1. Project Tasks and Schedule

[illegible]

The figure on the previous page ([available online](#)) depicts Gantt's diagram, outlining an ordered sequence of tasks, phases and milestones of the project. It graphically projects the estimate of subsequent tasks and may become a subject to change and adjustments as the project progresses to adapt to the unforeseen circumstances. It is worth-mentioning that several tasks tend to overlap and some, in the temporal sense, span beyond the delimited phases. This is mainly to address the requirement concerns mentioned in the assignment instructions (design phase deliverables) and to better incorporate agile methodologies used. For the latter the team members utilize a mixture of Extreme programming and Pair programming.

4.2. Continuous Integration, Delivery and Deployment Plan

The strategy of the team for the Continuous Integration & Continuous Delivery (CI/CD) involves the combination of GIT branching strategy and utilization of GitLab Pipelines.

As for the GIT branching strategy we use the following branch structure:

- Master
 - Dev
 - Feature

The `Master` branch contains only the tested peer-reviewed code that is merged from a `Dev` branch using GitLab's Merge Request functionality (equivalent to GitHub's Pull Request). The `master` branch is de facto equivalent to the *release* of a tested product. For each novel piece of functionality that will be implemented, the responsible team member creates a `Feature` branch named with the syntax similar to the following: `feature/#01developerName_name-of-issue` where the number after the hash symbol `#` is linked with the issue number in GitLab. GitLab automatically maps pushed changes to the corresponding issue numbers, making it easy to track the changes in the user interface of the GitLab environment.

As for the process of merging of feature branches to the `Dev` and `Master` branches a simple procedure was agreed upon. For each merge request that a team member makes, he/she must assign a reviewer and a deadline to which the code addition should be reviewed. Reviewer has to be a team member other than the developer making the request. The responsible reviewer will review, approve and merge the changes, respectively. In case that the reviewer finds the merge request inadequate he will address the issues found by commenting in the respective merge request. In this context, the most important rule is that a team member that has made a merge request **must not** merge the changes by him/herself but rather must always assign the accountability to another team member who is thereby responsible for approved changes.

The following sequence describes the CI/CD workflow:

1. CI/CD executes maven build steps which will includes operation in the following order:
 - a. (Optional) Pre-build static code analysis
 - b. Build applications
 - c. unit tests
 - d. integration tests
 - e. (Optional) Quality Gate
 - f. Create Docker Image
 - g. Deploy Image

Additionally, for manual deployment of each Microservice, it will be possible to create a self-contained `.tar` file. Moreover, to enable integration testing of a given Microservice, in-memory database will be implemented.

4.3. Distribution of Work and Efforts

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:	<ul style="list-style-type: none"> • SR3 • SR8
Code Ownership	<ul style="list-style-type: none"> • Inventory Product MS • Shop Front-end
Design	<ul style="list-style-type: none"> • Gantt Chart • Inventory Product MS classes • Formulation of the chapter 4. • Product Inventory Mocks • Front-end • Use-cases

Contribution of *Puri Himal*:

Scope requirements responsibility:	<ul style="list-style-type: none"> • SR6 • SR7
Code Ownership	<ul style="list-style-type: none"> • User MS
Design	<ul style="list-style-type: none"> • Sequence Diagrams • Use-cases

Contribution of *Preisinger Johannes*:

Scope requirements responsibility:	<ul style="list-style-type: none"> • SR4 • SR5
Code Ownership	<ul style="list-style-type: none"> • Payment Transaction MS • Authentication MS • BFF MS
Design	<ul style="list-style-type: none"> • Deployment Diagram • Formulation of chapter Design Overview • Continuous Integration, Delivery and Deployment Plan • Development Stack and Technology Stack • Use-cases

Contribution of *Lascsak Christian*:

Scope requirements responsibility:	<ul style="list-style-type: none"> • SR9 • SR11
Code Ownership	<ul style="list-style-type: none"> • Notification MS
Design	<ul style="list-style-type: none"> • Component Diagram • Formulation of chapter Design Decisions • Notification classes • Use-cases

Contribution of *Kapeller Angelika*:

Scope requirements responsibility:	<ul style="list-style-type: none"> • SR10 • SR12
Code Ownership	<ul style="list-style-type: none"> • Shipment MS
Design	<ul style="list-style-type: none"> • Use-case Diagram • Formulation of chapter Design Assumptions • Shipment Classes • Use-cases

Team effort time estimates

Date	Hours	Done	Name
10.03.2021	2	Getting started with the technologies required. Setup of simple docker framework and management scripts	Johannes Preisinger
15.03.2021	1	Provide first draft of Create a Deployment Diagram	ALL
17.03.2021	1	Create Use Case Diagram	Angelika Kapeller
18.03.2021	3	Project Kick-off, use case diagram discussion	ALL
25.03.2021	3	Deployment diagram and use case diagram discussion, agree on branching strategy	ALL
01.04.2021	4	sequence diagram, component diagram, use case diagram discussion	ALL
01.04.2021	2,5	Create textual description of use cases	Angelika Kapeller
05.04.2021	3	Auth Microservice	Johannes Preisinger
06.04.2021	3	Sequence diagram draft	Puri Himal
08.04.2021	3	Create Component Diagram	Christian Lascsak
08.04.2021	3,5	Discussion of use case textual description, sequence diagram, component diagram; agree on work distribution for design report	ALL
08.04.2021	1,5	Assigned Use-cases Review and expansion	Christian Lascsak
08.04.2021	1	Assigned Use-cases Review and expansion	Puri Himal
08.04.2021	1,5	Assigned Use-cases Review and expansion	Hynek Zemanec

08.04.2021	1	Assigned Use-cases Review and expansion	Johannes Preisinger
08.04.2021	1	Assigned Use-cases Review and expansion	Angelika Kapeller
08.04.2021	2,5	Sequence diagram modification and expansion	Puri Himal
09.04.2021	1,5	Discuss solutions for event-driven design approach, look at implications for diagrams	ALL
10.04.2021	1	Kafka Setup	Johannes Preisinger
11.04.2021	2,5	Update Component Diagram for Async Comm.	Christian Lascsak
12.04.2021	1	Class Diagram Auth	Johannes Preisinger
12.04.2021	1	Class Diagram Inventory Service	Hynek Zemanec
12.04.2021	3	Sequence diagram change	Himal Puri
13.04.2021	2,5	Distribute code ownership for MS and SR, discuss Kafka setup within sequence and component diagram	ALL
15.04.2021	1	Kafka Message Example	Johannes Preisinger
16.04.2021	0,5	Update Component Diagram	Christian Lascsak
16.04.2021	0,5	Send mock objects over Kafka	Johannes Preisinger
16.04.2021	3	Discuss project setup of MS with Docker, Springboot and Kafka	ALL
16.04.2021	2,5	Front-end concept design (HTML)	Hynek Zemanec
17.04.2021	0,5	Update Component Diagram	Christian Lascsak
18.04.2021	2	Setup Notification Microservice	Christian Lascsak
18.04.2021	1	Short introduction into how to set up MSs	Johannes Preisinger
18.04.2021	2,5	Setup (former) Vendor Microservice	Angelika Kapeller
19.04.2021	1	Class Diagram Transaction	Johannes Preisinger
19.04.2021	1	Class Diagram for Notification Microservice	Christian Lascsak
19.04.2021	1	Class Diagram for Product Service	Himal Puri
20.04.2021	2,5	Formulation of chapter 4	Hynek Zemanec
21.04.2021	2	Guide through the microservice Layering	Johannes Preisinger
21.04.2021	2,5	Review setup of MS	ALL
22.04.2021	2	Writing about the development stack, technology stack and non functional requirements	Johannes Preisinger
22.04.2021	4	Front-end React routing, data fetching	Hynek Zemanec
23.04.2021	3	Design Discussions and Iterating	ALL

		through the solution	
22.04.2021	3,5	Mockable Mocks	Christian Lascsak
23.04.2021	4,5	MS User setup	Puri Himel
23.04.2021	6,5	MS Inventory Setup + Mock	Hynek Zemanec
23.04.2021	3,5	Technical Support for MS setups	Johannes Preisinger
23.04.2021	2,5	Formulation of Chapter 1	Angelika Kapeller
23.04.2021	1	Class Diagram for Shipment Microservice	Angelika Kapeller
24.04.2021	2	Finalize Mockups transaction service	Johannes Preisinger
24.04.2021	5	Add Mockup for notification Service, update component diagram, update notification class diagram, update software design, update Design decisions chapter	Christian Lascsak
24.04.2021	1.5	Make Kafka available for local dev, introduce H2 DB for local dev and testing	Johannes Preisinger
25.04.2021	3	Update Design Decision Chapter, update Notification Class Diagram, add System requirements	Christian Lascsak
25.04.2021	3	Work on report	ALL
25.04.2021	2	Write assumptions part	Angelika Kapeller
25.04.2021	4,5	Create shipment MS	Angelika Kapeller

5. How-to / mock-up documentation

[Mockable](#) for getters and setters - except those that publish information to the event store.

The event driven mocks can be executed via REST, as soon as the initial setup is started. The local setup can be started by executing `startup_local.sh`. Then the microservice environment will be started with Docker.

https://git01lab.cs.univie.ac.at/vu-advanced-software-engineering/students/2021s/ASE_Team_0501/-/issues/25

We employed <https://www.uuidgenerator.net/> to generate IDs.

We agreed to use the following product ids in our mocks to ensure consistency:

- 321f737d-f8dd-4c9d-aca8-b8c6164a1dd1
- 4c68e51d-04d5-4c3d-b943-79e7fcaea92f
- 8de5155a-e36b-4b3b-83d2-631e07df9078
- dfc6d5d8-3d15-4d8a-a29a-01f5894e122e

We also agree to use following transaction ids:

- 008865a4-8348-11eb-8dcd-0242ac130003
- 852f4261-4d36-4241-83cb-071b37a4ff95
- 82b01891-3577-4192-8bd7-71bdc0781acb
- ba526480-3eea-4c3a-95cc-a0909de129bd

Following seller ids:

- 6d8ea0ea-1162-4739-87b4-6729ef7f1a20
- 5a48c43f-4c24-4ee5-bf75-430f83153349

Following buyer ids:

- 6d8ea0ea-1162-4739-87b4-6729ef7f1a20
- 5a48c43f-4c24-4ee5-bf75-430f83153349

Payment Transaction Service

The payment transaction service offers following mocks

POST /api/v1/transactions

will transmit a payment transaction mock to the event store and every user interested in that event can listen to it

```
r ID: Xhg8pyMGRAGGeR4aJaB00g
transaction-service_1 | 2021-04-24 12:50:32.240 INFO 1 --- [ntainer#0-0-C-1]
a.e.s.s.t.services.TransactionService : {"uuid":"008865a4-8348-11eb-8dcd-0242
ac130003","seller":"5a48c43f-4c24-4ee5-bf75-430f83153349","buyer":"6d8ea0ea-1162
-4739-87b4-6729ef7f1a20","currency":"EUR","createdAt":"-999999999-01-01T00:00:00
","modifiedAt":"+999999999-12-31T23:59:59.999999999","productIds":null,"address"
:{"firstName":"Johannes","lastName":"Preisinger","email":"cnnwarthem@gmail.com",
"street":"Cumberlandstraße","streetNumber":"111","doorNumber":"7","floor":"1. Ke
llergeschoss","city":"Vienna","zipCode":"1140","country":"AT"},"paymentMethod":"
PAYMENT_ON_DELIVERY","products":[{"id":"321f737d-f8dd-4c9d-aca8-b8c6164a1dd1","p
rice":29.21},{id":"4c68e51d-04d5-4c3d-b943-79e7fcaea92f","price":10021.23},{id
":"8de5155a-e36b-4b3b-83d2-631e07df9078","price":10.0},{id":"dfc6d5d8-3d15-4d8a
-a29a-01f5894e122e","price":22.21}]}
transaction-service_1 | 2021-04-24 12:50:32.280 INFO 1 --- [ntainer#0-0-C-1]
```

The domain transaction also offers three GET services for clients, that are provided by mockable.

Get all system transactions

GET /api/v1/transactions

<https://demo3220613.mockable.io/api/v1/transactions>

Calculate profits for a specific vendor/seller

GET api/v1/transactions/sellers/5a48c43f-4c24-4ee5-bf75-430f83153349/profits

<https://demo3220613.mockable.io/api/v1/transactions/sellers/5a48c43f-4c24-4ee5-bf75-430f83153349/profits>

Get all transactions filtered for a specific user and a specific role

GET api/v1/transactions%3FuserId=5a48c43f-4c24-4ee5-bf75-430f83153349&role=BUYER

<https://demo3220613.mockable.io/api/v1/transactions%3FuserId=5a48c43f-4c24-4ee5-bf75-430f83153349&role=BUYER>

We also want to allow additional filter parameters, like filtering the transactions according to date. This feature will be added to the final implementation.

Notification Service

The notification service offers the following mock:

POST `api/v1/productmarking`

<https://demo7574143.mockable.io/api/v1/productmarking>

Mockable:

Path

Verb

☐ GET ☒ POST ☐ PUT ☐ PATCH ☐ DELETE ☐ OPTIONS
Select the HTTP verb for this mock. If you select OPTIONS, the CORS standard will be disabled for all mocks running on the same path.

headers

+

status

content-type

Choose

application/json

Content-Type Header for response

encoding

UTF-8

Content-Encoding Header for response

body

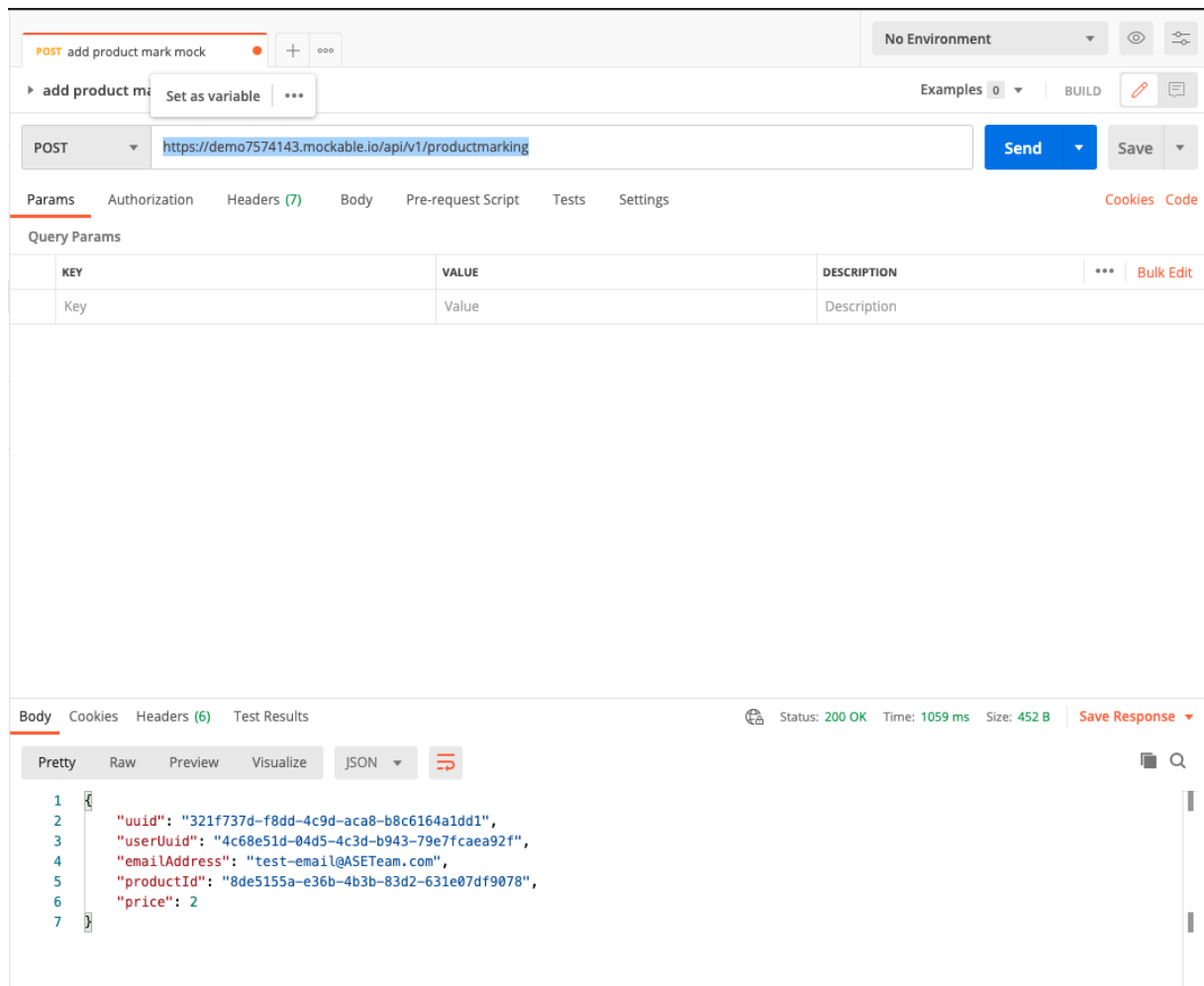
Enable Dynamic Response: ☐

1- {
2 "uid": "321f737d-f8dd-4c9d-aca8-b8c6164a1dd1",
3 "userId": "4c68e51d-04d5-4c3d-b943-79e7fcae092f",
4 "emailAddress": "test-email@ASET.com",
5 "productId": "8de5155a-e36b-4b3b-83d2-631e07df9078",
6 "price": 2
7 }
8

name

Optional. We will generate one for you based on the Verb and Path.

description

Postman call:

Product Inventory Microservice

The Product Inventory service offers the following 9 mocks using the service [mockable.io](https://demo7574143.mockable.io).

All Products

GET `api/v1/products`

- <http://demo1469845.mockable.io/api/v1/products>

Product

GET `api/v1/products/{productID}`

- <http://demo1469845.mockable.io/api/v1/products/321f737d-f8dd-4c9d-aca8-b8c6164a1dd1>

- <https://www.mockable.io/a/#/space/demo1469845/rest/RNEc-GAAA>
- <http://demo1469845.mockable.io/api/v1/products/8de5155a-e36b-4b3b-83d2-631e07df9078>
- <http://demo1469845.mockable.io/api/v1/products/4c68e51d-04d5-4c3d-b943-79e7fcaea92f>

Vendor's inventory

GET `api/v1/inventory/{vendorID}`

- <http://demo1469845.mockable.io/api/v1/inventory/c64eee9c-a4dc-11eb-bcbc-0242ac130002>

Create Product

POST `api/v1/products/{vendorID}`

- <http://demo1469845.mockable.io/api/v1/products/c64eee9c-a4dc-11eb-bcbc-0242ac130002>

Update Product

PUT `api/v1/products/{vendorID}`

- <http://demo1469845.mockable.io/api/v1/products/321f737d-f8dd-4c9d-aca8-b8c6164a1dd1>

Update Inventory

PUT `api/v1/inventory/{vendorID}`

- <http://demo1469845.mockable.io/api/v1/inventory/c64eee9c-a4dc-11eb-bcbc-0242ac130002>

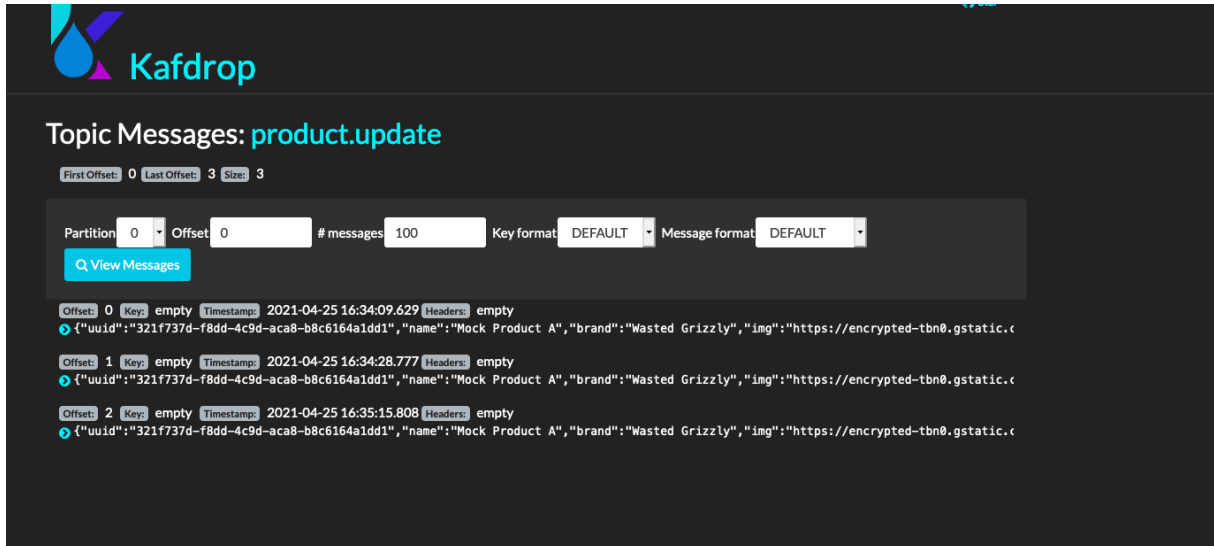
product.update

To trigger the kafka publish call:

POST <http://localhost:4008/api/v1/inventory>

Now go to kafdrop at <http://localhost:9000>

navigate to the topic `product.update > messages`. There you can see the mocked message sent by the product inventory service (see the image below).



The screenshot shows the Kafdrop web interface. At the top, the Kafdrop logo is visible. Below it, the title 'Topic Messages: product.update' is displayed. There are filters for 'First Offset: 0', 'Last Offset: 3', and 'Size: 3'. A search bar with 'Q View Messages' is present. Below the search bar, there are input fields for 'Partition: 0', 'Offset: 0', '# messages: 100', 'Key format: DEFAULT', and 'Message format: DEFAULT'. The main content area displays three messages, each with its offset, key, timestamp, headers, and a JSON body. The messages are for 'Mock Product A' with a 'brand' of 'Wasted Grizzly' and an 'img' URL.

Shipment Microservice

The shipment service offers following mocks

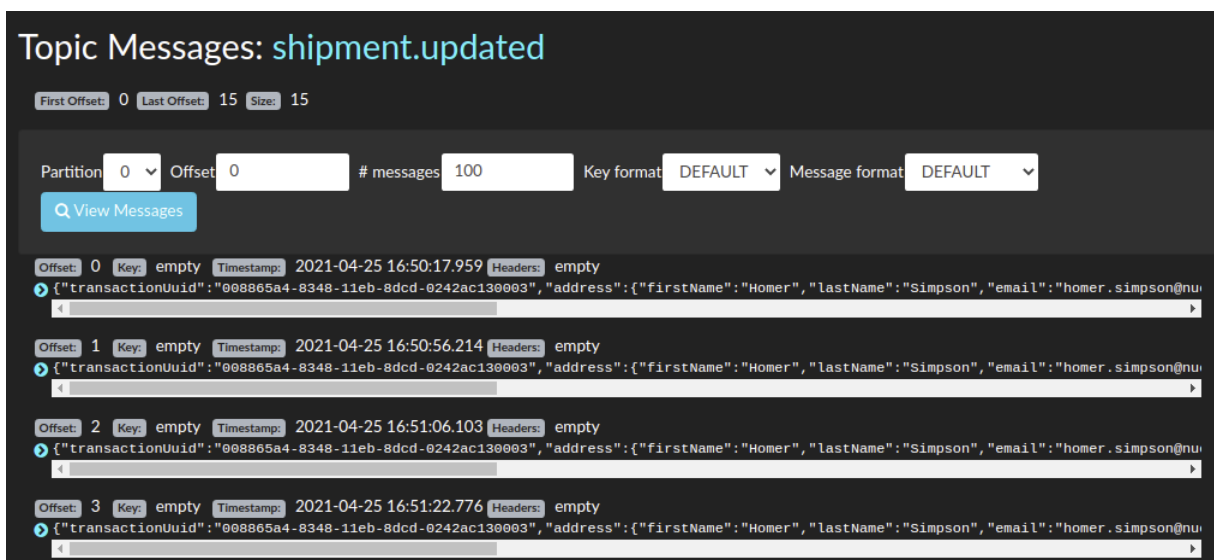
shipment.update

To trigger the kafka publish call:

POST <http://localhost:4007/api/v1/inventory>

Now go to kafdrop at <http://localhost:9000>

navigate to the topic shipment.update > messages. There you can see the mocked message sent by the shipment service.



The screenshot shows the Kafdrop web interface for the 'shipment.updated' topic. The title is 'Topic Messages: shipment.updated'. The filters show 'First Offset: 0', 'Last Offset: 15', and 'Size: 15'. The search bar has 'Q View Messages'. The input fields for 'Partition: 0', 'Offset: 0', '# messages: 100', 'Key format: DEFAULT', and 'Message format: DEFAULT' are visible. The main content area displays four messages, each with its offset, key, timestamp, headers, and a JSON body. The messages contain a 'transactionUuid' and an 'address' object with 'firstName', 'lastName', and 'email' fields.

User MS

The user MS offers five mockups. it was done via mockable and as well as in implementation. For implementation, it uses a collection to store the information for the mockups. During the final phase this MS will have its own database.

Create user with product: POST

<https://demo7418841.mockable.io/api/v1/users>

Get user with userid: GET

<https://demo7418841.mockable.io/api/v1/users/321f737d-f8dd-4c9d-aca8-b8c6164a1dd1>

Getting cart of user with userid: GET

<https://demo7418841.mockable.io/api/v1/users/321f737d-f8dd-4c9d-aca8-b8c6164a1dd1/cart>

Adding product to user's cart: POST

<https://demo7418841.mockable.io/api/v1/users/321f737d-f8dd-4c9d-aca8-b8c6164a1dd1/cart/products>

Delete product from user:DELETE

<https://demo7418841.mockable.io/api/v1/users/%20321f737d-f8dd-4c9d-aca8-b8c6164a1dd1/cart/product/%20321f737d-f8dd-4c9d-aca8-b8c6164a1dd3>