Advanced Software Engineering

DESIGN REPORT

Team number:	0105

Team member 1	
Name:	Kevin Miller
Student ID:	01638797
E-mail address:	a01638797@unet.univie.ac.at

Team member 2	
Name:	Lukas Sorer
Student ID:	01250600
E-mail address:	a01250600@unet.univie.ac.at

Team member 3	
Name:	Tobias Fühles
Student ID:	11936325
E-mail address:	a11936325@unet.univie.ac.at

Team member 4	
Name:	Dorina Karameti
Student ID:	01528726
E-mail address:	a01528726@unet.univie.ac.at

Team member 5	
Name:	David Cömert
Student ID:	01225308
E-mail address:	a01225308@unet.univie.ac.at

1. Design Draft

1.1. Design Approach and Overview

1.1.1. Assumptions

General Assumptions

- The systems that the services are deployed to have stable internet connection and are capable of accepting and sending HTTP requests
- The services have to not be containerized and easily deployable for DESIGN
- No testing (aside from basic smoke testing) was required for DESIGN
- The user has access to a web browser to interact with the applications frontend

Users

- The shop has a set user base with a set amount of accounts, the user can not create or modify account data manually
- The user has no expectation of actual profit
- The user has no expectation of selling a real item
- The user has no expectation of buying a real item
- The user knows that the items displayed are not items that actually exist and are for sale
- The user does not expect to receive an actual physical shipment if they bought something in the store
- The user knows that this application was designed as a semester project by students and expects to encounter potential flaws in the system

Shop

Some assumptions coincide with the user assumptions and will not be duplicated (e.g. you can not really sell an item, no profit will be paid out, etc.)

- The shop will not be used to sell and buy actual physical or digital products
- The shop's content (available items, current shipments) can be created manually by the development team and does not have to be based on real world occurrences
- In general, all activity in the shop does not reflect any real world activity and does not trigger any real world actions

1.1.2. Design Decisions

Decision 1 - Spring Boot

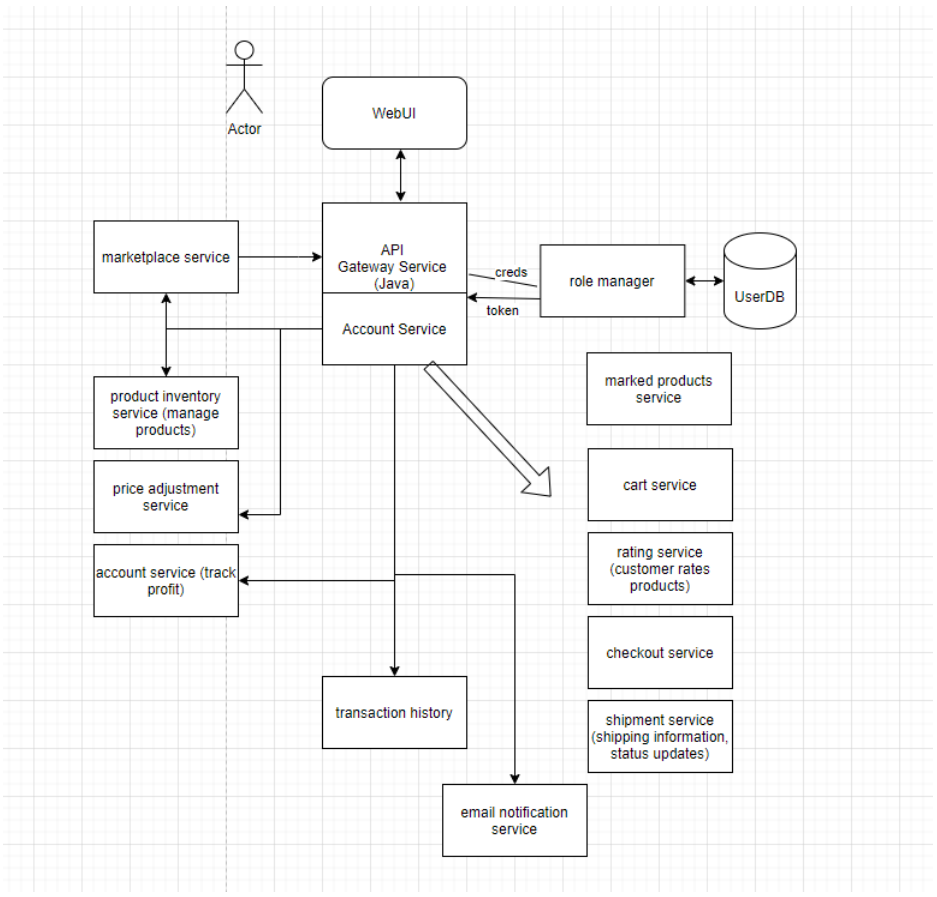
One could argue that choosing a framework is not a design decision, but this choice impacted our architecture in many ways. Spring Boot allowed us to make the architecture a lot simpler by omitting a lot of base-infrastructure topics that would have otherwise been of great concern. Spring Boot is a framework that perfectly fits the scope of this project. It would be a handicap not to use it in our opinion. This decision was not really up to debate as soon as we read that the recommended language for development was Java and it was finalized instantly without debate.

Decision 2 - The Gateway

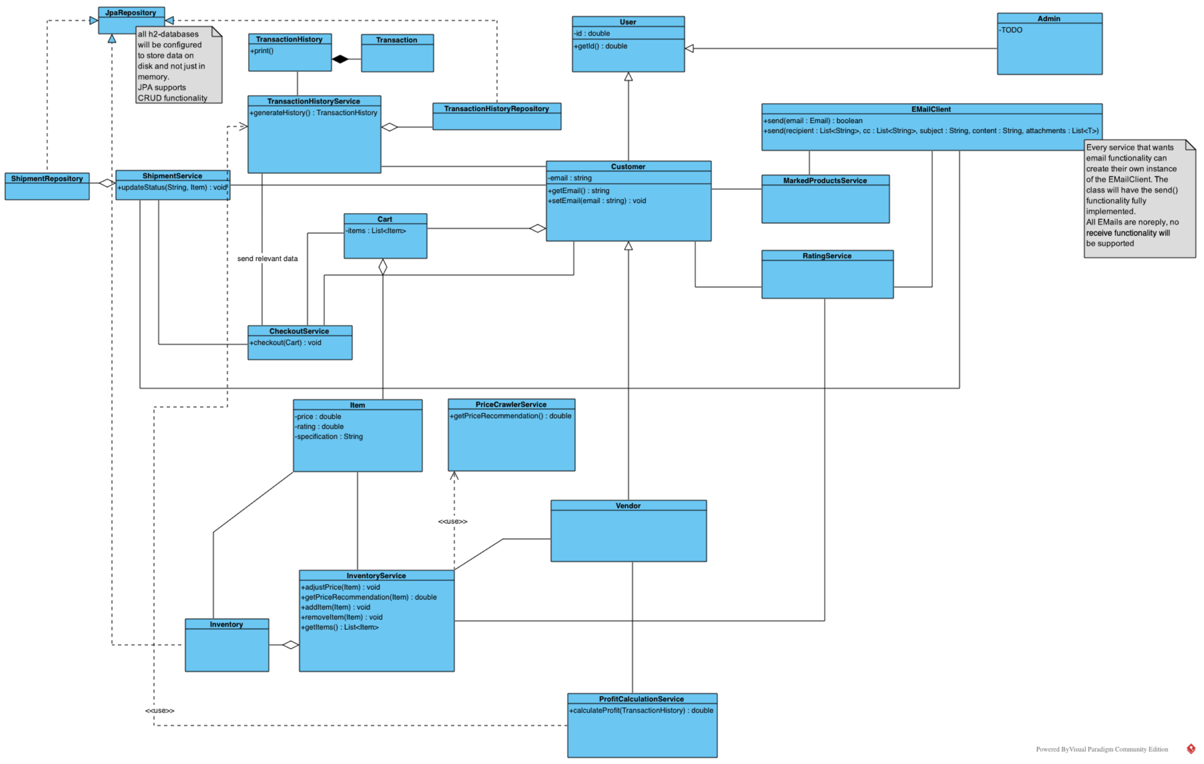
After doing some basic research on existing webshop architectures, we quickly figured that a gateway based solution was the way we wanted to go. The gateway acts as a single point of contact to the web based frontend, that further distributes all requests to the individual services. To reduce redundant text in this report, a much more detailed explanation of this decision and its impact on our project can be found in the architectural overview.

Decision 3 - The Gateway as role manager

It took some iterations to decide on the gateway based solution (see Step 3 Design Overview). We think that we were too focussed on the roles, which caused us to think in a much more complicated way than necessary. In this first initial draft we incorporated an Account Service, that should act as a second proxy layer (aside from the gateway) from the user to the actual services.


Actor
WebUI
API Gateway Service (Java)
Account Service
marketplace service
role manager
creds
token
UserDB
product inventory service (manage products)
price adjustment service
account service (track profit)
transaction history
email notification service
marked products service
cart service
rating service (customer rates products)
checkout service
shipment service (shipping information, status updates)

This architecture was omitted rather quickly, because all of the desired functionality could simply be incorporated right into the gateway instead.


JpaRepository
all h2-databases will be configured to store data on disk and not just in memory. JPA supports CRUD functionality
TransactionHistory
+print()
Transaction
TransactionHistoryService
+generateHistory() : TransactionHistory
TransactionHistoryRepository
ShipmentRepository
ShipmentService
+updateStatus(String, Item) : void
Cart
-items : List<Item>
send relevant data
CheckoutService
+checkout(Cart) : void
User
-id : double
+getId() : double
Admin
-TODO
EMailClient
+send(email : Email) : boolean
+send(recipient : List<String>, cc : List<String>, subject : String, content : String, attachments : List<T>)
Every service that wants email functionality can create their own instance of the EMailClient. The class will have the send() functionality fully implemented. All EMails are noreply, no receive functionality will be supported
Customer
-email : string
+getEmail() : string
+setEmail(email : string) : void
MarkedProductsService
RatingService
Item
-price : double
-rating : double
-specification : String
PriceCrawlerService
+getPriceRecommendation() : double
<<use>>
Vendor
InventoryService
+adjustPrice(Item) : void
+getPriceRecommendation(Item) : double
+addItem(Item) : void
+removeItem(Item) : void
+getItems() : List<Item>
Inventory
ProfitCalculationService
+calculateProfit(TransactionHistory) : double
<<use>>
Powered By Visual Paradigm Community Edition

We then took a step backward and thought about a solution that doesn't require any gateway or central solution at all. We came up with the idea that the services will be bound to user objects, that sort of aggregate connection details and access permissions to the individual

services. We even tried to design a full model of this, but after a couple of days of using this model as the base for further development, we just couldn't think of a scenario where this design would be in any way superior to the gateway-based solution.

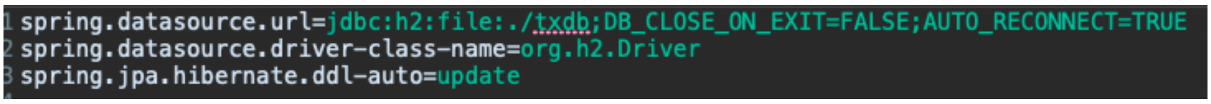
We then tossed everything overboard and started from scratch, knowing that we do in fact require a central solution, as we never thought about where these user-objects would actually exist (in memory in some service, as their own service etc.). We tried to maintain as much of the architecture from our first draft, but tried simplifying it as much as possible and ended up back at the gateway based solution. The gateway will also take care of all user/account related functionality (like selecting what requests can be executed based on role).

Decision 4 - Publish & Subscribe

Initially we wanted to hard-wire the service addresses into the gateway via a properties file in the gateway, which would contain information about where the gateway could reach the services. This decision stuck around for a surprisingly long time given that we knew about the publish-subscribe requirement and also given how obviously inferior this design is compared to registering and subscribing. As will be mentioned in the design overview, the services register with the gateway (the registration message contains their category, available endpoints and their ip+port) in order to be eligible to be forwarded user requests. The original design was that every service category had its own line in a .properties file, that would be read by the gateway whenever it was necessary to forward a request to that specific service category.

Decision 5 - JPA Repository vs. Database

Theoretically a JPA Repository is also considered a database, but initially we wanted each service to have their own document based database (like MongoDB). We never considered an SQL based solution, as it just seemed bloated, but it also would've been a possibility. We then stumbled upon an article that covered how to make JPA repositories exist beyond runtime, where the configuration was surprisingly easy. All that was necessary was to add the following lines to the application.properties file in the respective services that required permanent storage:



```
spring.datasource.url=jdbc:h2:file:./txdb;DB_CLOSE_ON_EXIT=FALSE;AUTO_RECONNECT=TRUE
spring.datasource.driver-class-name=org.h2.Driver
spring.jpa.hibernate.ddl-auto=update
```

It then creates a .db file in the service's root directory, that will be reloaded upon restarting. This allows for all the functionality we wanted regarding permanent storage beyond runtime, so we went with it. The nice integration of the Java Persistence API into Spring Boot, like autowiring the repository and the easy import of the h2 driver through Maven, sealed the deal for this decision rather quickly.

What's important to mention is that this solution makes it mandatory for the service to be developed in Java. A decision that was fine for all members, but might not be fine for others. However, since this is a microservice based project, if a service was to be developed in another language, it could simply use its own database solution without impacting interoperability whatsoever.

Decision 6 - No shared data classes

This decision plagued us for quite a long time, but we finally decided that we did not want to go the route of a separate project that is used by all services and contains data classes.

To elaborate, we thought about extracting all data objects/jpa entities that are commonly shared over the network (e.g. Item, Transaction, Shipment, etc.) into a separate project that would be imported by all services. Since Spring Boot supports the automatic marshalling of POJO's (returning a POJO from a @RequestMapping annotated function transforms it into JSON/XML and back into POJO if just annotated as "@RequestBody"), an amazing feature by the way, this would make sending the data objects as easy as passing them as a parameter to any other local function.

For the sake of preserving the core idea of microservices we decided against it, as it would further cement the decision of using Java as our only development language. It would certainly make development easier, but also - in our opinion - miss the point of the project.

Therefore, we rely on API specification in order to create and marshal data objects that are unique to the respective service.

Decision 7 - Frontend

To serve a basic Frontend based on HTML we choose a NodeJS web server. NodeJS is highly optimized to handle multiple network connections and have a nonblocking I/O. This handles all the routes and rearranges the format of the data from the user input. We decided to use this server as a microservice. On the frontend we distinguish between customer, vendor and administrator. Each of them has its own view to fulfill their use cases. With the Express-framework in NodeJs this can easily be realized. Additionally we decided to use JSON as our message format to communicate between all of the micro services. The benefit now for NodeJs is the strong similarity between JavaScript objects and JSON, which can be seen as a kind of serialization of the Javascript object. To program is type safe, the source code is written in Typescript and it is transpiled to JavaScript.

Decision 8 - Login

After properly reading through the semester project requirements we found no explicit duty to implement any complex login functionality. As the semester project assignment paper states:

> *"For reasons of data protection and privacy (see GDPR2) a user is not allowed to access any part of the system that is not directly pertinent to her or his role or use-case"*

We decided as a team to fulfill the correct distribution of access rights by a unique identifier i.e., the userId. This userId can distinguish if a given user is a customer, a vendor or an admin user role. We agreed to take this into account as we build our test cases and save user data into our databases.

1.1.3. Design Overview

Architectural Overview

This segment is intended to serve as a general overview of our design and to summarize the design decisions into a compact segment. Some information may be duplicate.

We wanted to separate all requested Scope Requirements into highly cohesive and self contained microservices that contain all the necessary logic to execute their job and their job only. These jobs are then orchestrated and distributed by the **gateway**

service. The gateway service acts as the single point of contact to the user and the frontend application (webpage).

All actions the user desires to make (e.g. add items to cart, check out cart, track shipments etc.) are executed via REST-calls to the gateway's according endpoints. All necessary information is bundled in the request's body by the frontend and forwarded to the gateway, which (if necessary) manipulates the data and forwards or orchestrates the action to be dealt with by the microservices.

In order for these microservices to be known to the gateway, they have to register with it. This is done via a simple HTTP-Post request from the service to the gateway that contains the information about the services *category* (these categories are based on the SR's, e.g. "inventory", "history" etc.), its *address* (ip + port) and the *endpoints* that are available. This information is then stored in the gateway within the *TrafficController*. This class is responsible for storing the registered services and for storing the entire network traffic to the gateway as *Message* objects. These message objects are stored in memory and queued in a MessageQueue style way. Whenever the message is passed on, it is deleted. This ensures delivery even if microservices are temporarily unavailable.

When an incoming request triggers an action (e.g. add an item to inventory), all services that are subscribed to the category "inventory" will be forwarded this request. This allows for theoretical redundancy. If the request triggers a more complex action (e.g. user wants to checkout cart), this action is orchestrated by the gateway. In this example, the gateway would fetch relevant cart data from a *cart* service, forward this data to a *checkout* service and forward the *checkout* service's response to the *history* and the *shipment* service.

This method allows for very easy extension and maintenance, as more functionality can be added by simply adding more services and extending the gateway to accommodate the added functionality. The gateway could also be redundant, but we chose to not incorporate this aspect into our implementation. We are also aware that this creates a theoretical single point of failure, this being the gateway. Since the gateway could be made redundant, we still deemed this approach to be the most valid design choice for this task. Also, the load-balancing architecture is very common for this domain (e.g. in Amazon). It is simply not feasible to create a robust architecture that still responds to requests if some components are unavailable without having a central entity (in this the gateway) for error management, message queueing and so forth. A diagram of this architecture can be found in chapter 3.2.

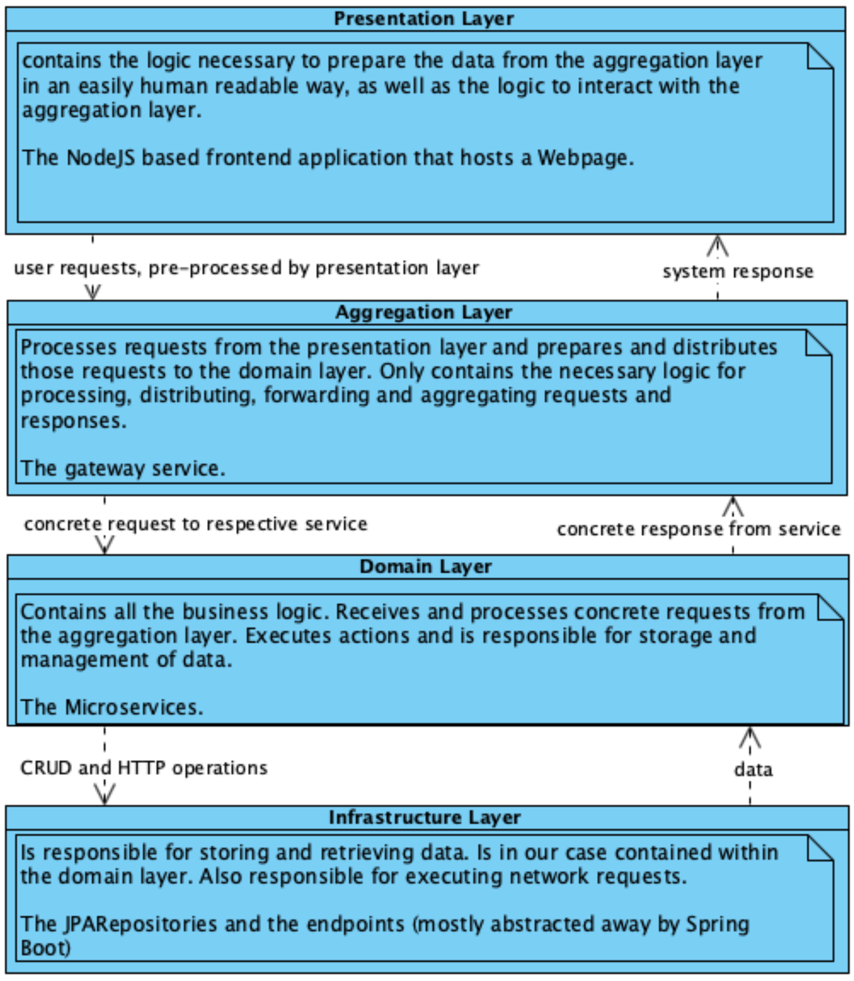
Design Patterns

The architecture of our implementation follows the principles of the classic l**ayered architecture**.

Our system is divided into 4 main layers:

- Presentation Layer; JavaScript based Webpage
- Aggregation Layer; Gateway
- Domain Layer; Microservices
- Infrastructure Layer; Persistent Storage and Networking

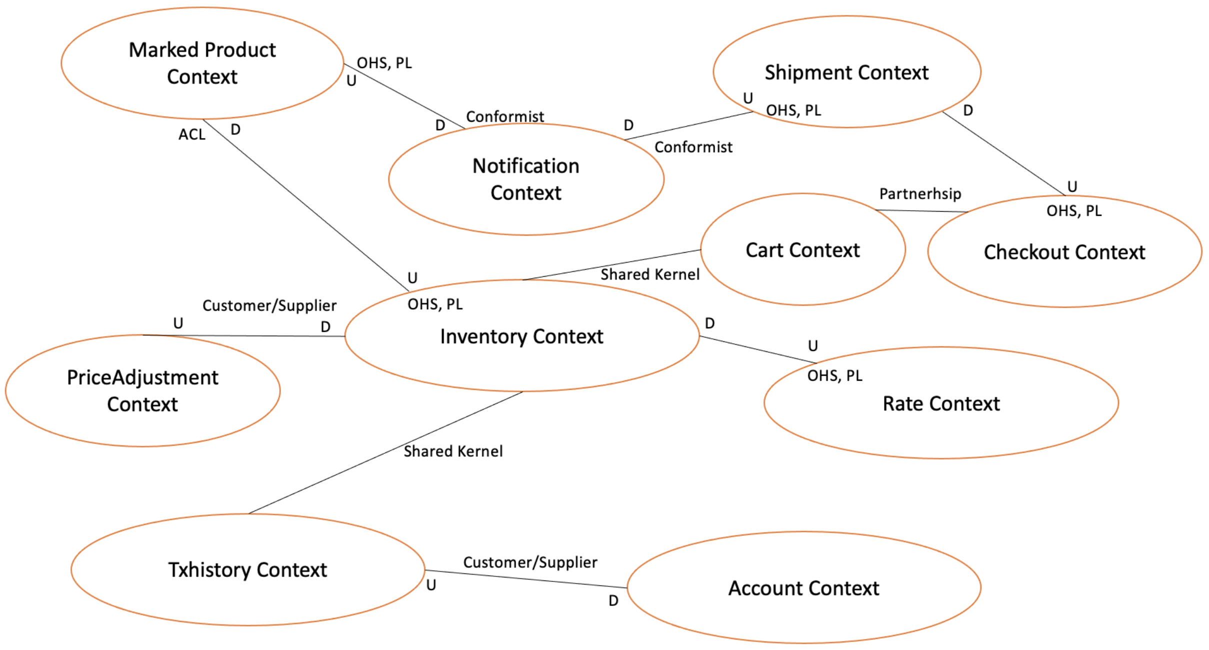
Thinking of our system as a layered architecture made the design process a lot simpler.


Presentation Layer
contains the logic necessary to prepare the data from the aggregation layer in an easily human readable way, as well as the logic to interact with the aggregation layer.
The NodeJS based frontend application that hosts a Webpage.
user requests, pre-processed by presentation layer
system response
Aggregation Layer
Processes requests from the presentation layer and prepares and distributes those requests to the domain layer. Only contains the necessary logic for processing, distributing, forwarding and aggregating requests and responses.
The gateway service.
concrete request to respective service
concrete response from service
Domain Layer
Contains all the business logic. Receives and processes concrete requests from the aggregation layer. Executes actions and is responsible for storage and management of data.
The Microservices.
CRUD and HTTP operations
data
Infrastructure Layer
Is responsible for storing and retrieving data. Is in our case contained within the domain layer. Also responsible for executing network requests.
The JPARepositories and the endpoints (mostly abstracted away by Spring Boot)

You could make the argument that Domain and Aggregation Layer are a single layer, but we deliberately chose to logically separate those system components into separate layers. It only made sense to us, as the Gateway service differs in so aspects compared to the other microservices. The Gateway is the only point of interaction to the frontend and - in our eyes - does not qualify as part of the domain layer, therefore it gets its own layer in the architecture.

Bounded Context Overview

Initially we discussed different options for partitioning the domain in bounded contexts. The first approach was led by naming and logical conventions, e.g. does an Item have identical meaning/implication in Inventory and Rating Context, and splitting the contexts accordingly. But soon after setting up the project structure it became clear that a mapping between most microservices and a bounded context seemed more reasonable. The following overview shows the context map, with our bounded contexts selected according to the domain, scope requirements and our project structure.

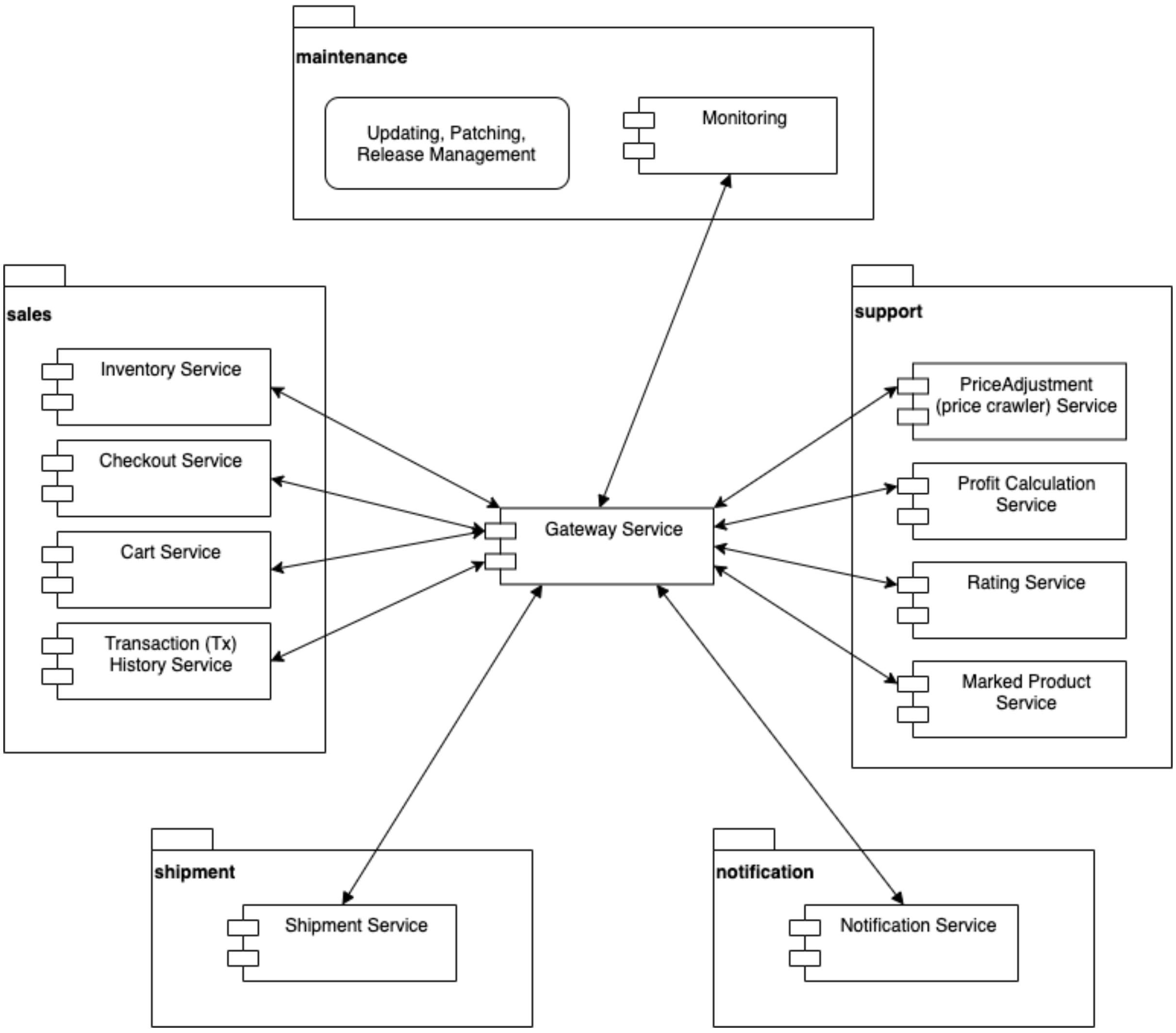


To take advantage of the microservice architecture with each team member working in separate bounded contexts, the above Context Map was created to set relationships and by that facilitating permanent considerations of dependent microservices. As all bounded contexts are developed internally the implementation of Anti-Corruption Layers was neglected as translation between contexts is monitored throughout the development process:

If your application needs to deal with a database or another application whose model is undesirable or inapplicable to the model you want within your own application, use an Anticorruption Layer to translate to/from that model and yours.

Notification Context was set as a Conformist to the Shipment as well to the Marked Product Context, as it serves them as a messenger and therefore should adhere its model to the suppliers. Further implementation will show if these assumption stands the test.

Logical Grouping into Language Contexts



To make working with a high number of bounded contexts easier, we decided to group some contexts into language contexts. You could also argue that these language contexts do not represent the actual bounded contexts, but further grouping the contexts made development and internal communication more efficient.

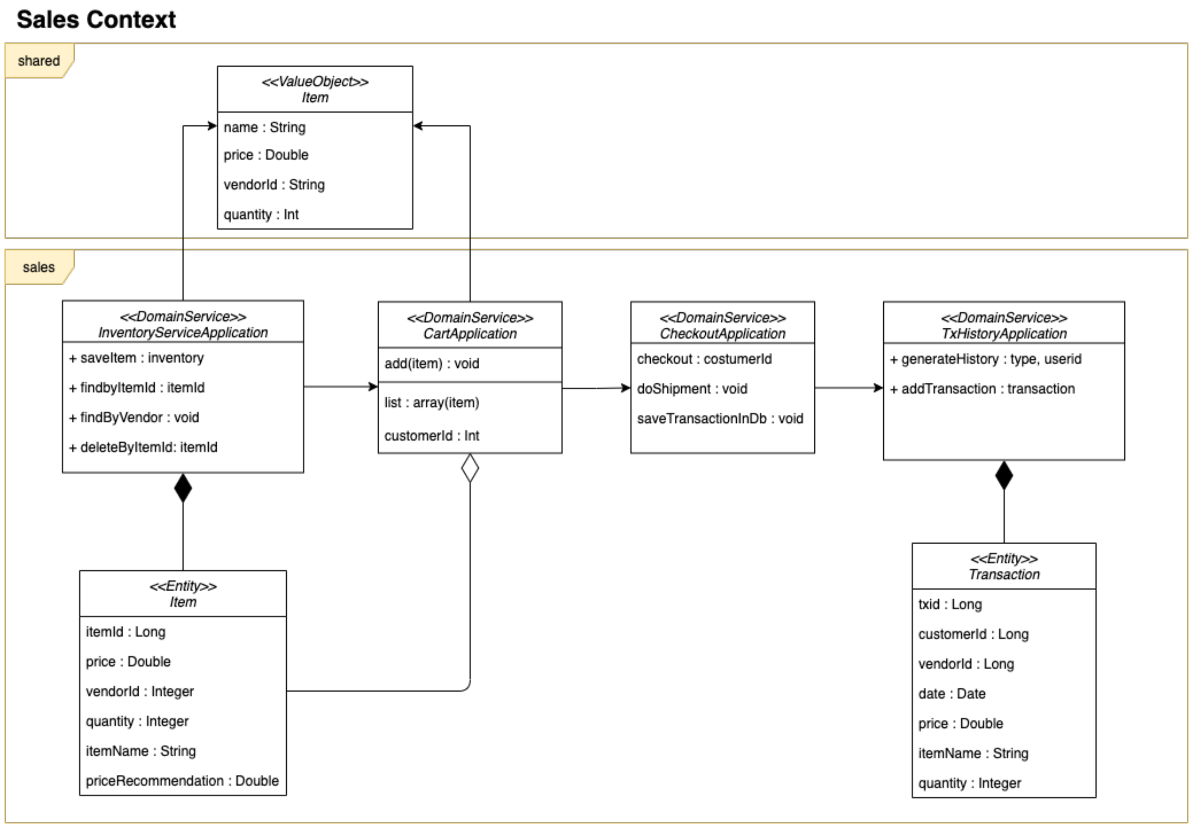
The diagram above showcases our view of the different language contexts of the web shop. Let us start on the left-hand side of the diagram with the sales context. This context has as its most important artifact the *item*, which can be used synonymously for a *product* that can be ordered by a customer. The main difference between the support-context, where there is also the concept of items and products, is that if we speak of items in the sales context, we think of them as objects that will be, already are or have been bought by a customer. One might also call this view the customer context as it is mainly focused with the customers perception of the web shop.

To the right we see the *support* context. This context is associated with functionality that boosts the user-experience (of both customer and vendor) when using the shop but lies outside the context of ordering and buying products. Here the concept of an item likewise plays an important role but rather as an object that has attributes that might change over the course of time. Let’s take the rating service as an example; the object that receives the rating is the item. This rating may play a role in the user perception of the item as a better rating might lead to a higher possibility that it is bought by the customer. Thus, it is just a functionality that makes the experience of shopping more convenient - it is distinguishable from an item that was actually bought.

We see two contexts that are somehow separated from the sales or support contexts as they form their own contexts. These contexts are the *shipment* and the *notification* context. Although the shipment of an item as such is an elementary part of a successful order-fulfillment its context differs from the sales context as the most important artefact is the *order*. Vice versa holds true for the notification context, here the artifact of highest interest is the *message* that is triggered and contains varying content in multiple stages of the order-fulfillment process.

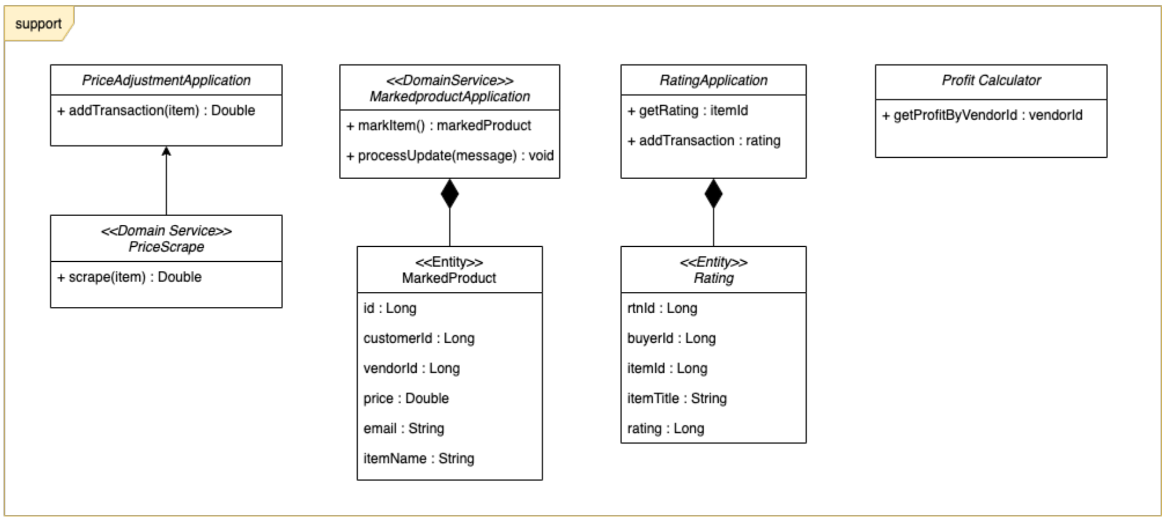
Finally we keep the *maintenance* context apart. This context is mainly driven by the point of view of an administrator that keeps the web shop from a technical standpoint alive.

It may be unnecessary to add the gateway service in this diagram as it does not have a context on its own. The reason we decided to put it in the diagram was to underline the importance of this element as it must know and understand the meaning of all the other contexts and streamline communication between these different aspects.



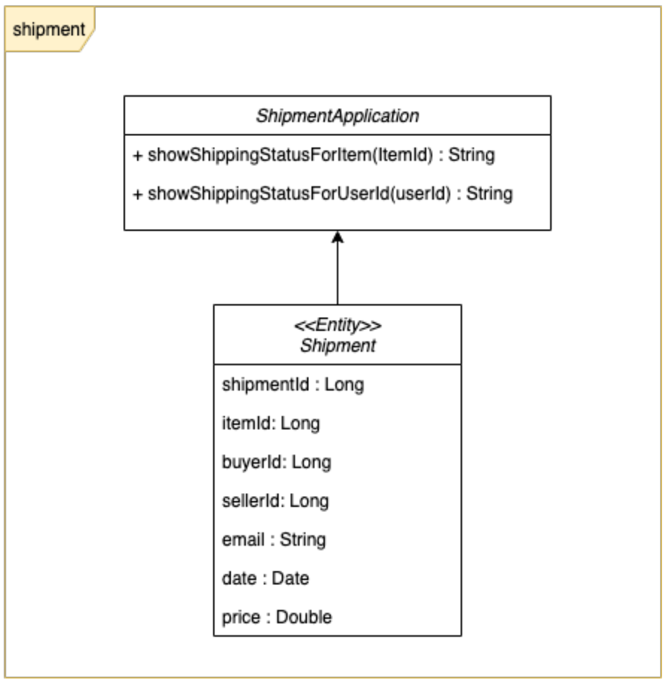
The sales context consists of one shared value object, four domain services and two entities. It includes the inventory service, which holds all items that user can buy, the cart (or shopping cart), which holds 0-n items that a user can add and order within one session, the checkout, which handles the correct proceeding in order to document and inform all relevant services that an order will arrive successfully to a customer and the transaction (tx or sometimes txn) history, which works as sort of logging facility that saves order information and provides this information to services along the delivery phase. The entity item is connected to the inventory service which gives an overview of all items of the shop (can be seen as some kind of a catalogue). The item is also connected with the cart as users can look through the inventory of the shop and choose which items they would like to buy. The entity transaction documents which customer has both how much from which vendor at a given time, how does the items are named and how high the price was at that point in time.

Support Context

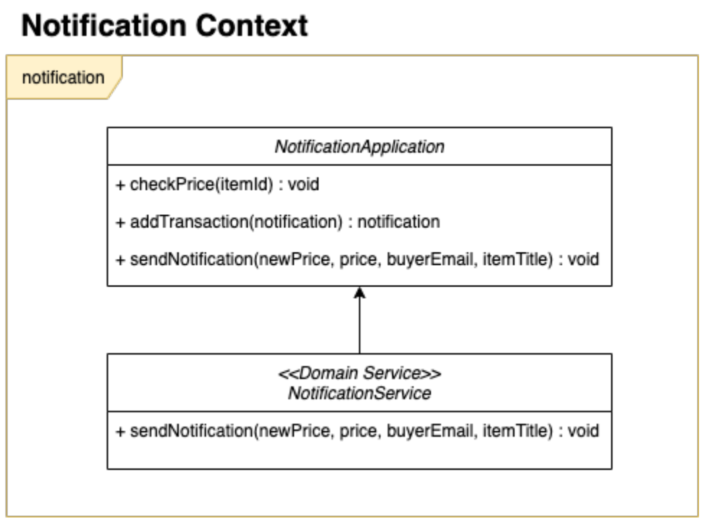


The support context consists of three applications, two domain services and two entities. Beginning from the left-hand side we see the price adjustment application that is connected to the domain service of price scrape. The domain service marked product is responsible for marking products and notifying (or at least triggering the notification service) customers or vendors if the price of an item has changed. The marked product service has a marked product entity attached to it that documents which customer has marked which product from which vendor. The rating application is responsible for rating items, the entity rating keeps track of these ratings. Lastly the profit calculator service is responsible for calculating the profit a vendor has realised at a given point in time.

Shipment Context



The shipment context summarizes the shipment application and the entity shipment. It is concerned with keeping track of the shipment status of a given order.


Notification Context
notification
NotificationApplication
+ checkPrice(itemId) : void
+ addTransaction(notification) : notification
+ sendNotification(newPrice, price, buyerEmail, itemTitle) : void
<<Domain Service>> NotificationService
+ sendNotification(newPrice, price, buyerEmail, itemTitle) : void

The notification context consists of the notification application and the notification service. It is responsible for customer notification when e.g. the status of an shipped order has changed.

1.2. Development Stack and Technology Stack

1.2.1. Development Stack

IDE

Eclipse (https://www.eclipse.org/downloads/)

IntelliJ (https://www.jetbrains.com/de-de/idea/)

Depending on preference and prior experience. We could not agree on a single IDE to be used as there are team members from both “camps”, that’s why we use both.

Build

Maven is awesome for managing dependencies and building projects. The team came to the decision that Maven is preferred over Gradle.

Typescript is transpiled into Javascript which runs directly in the NodeJs environment without building.

Deployment

Docker (https://www.docker.com/): Good containerization technology, ensures easy and reliable deployment.

Gitlab CI/CD (https://docs.gitlab.com/ee/ci/): ensures that everybody only commits high quality code.

1.2.2. Technology Stack

Programming Language

Java 11 (Java SE Development Kit 11- - Downloads): recommended by team and also a good fit for the project.

TypeScript 4.1 (https://www.typescriptlang.org/): Which gets compiled to JavaScript *(ES3) and runs on a NodeJs (https://nodejs.org/) environment.*

Frameworks

Spring-Boot (https://spring.io/projects/spring-framework): standard for Java Web-applications, previous experience of 1 team member

Spring Initializr (https://start.spring.io/): was used to generate the projects for all microservices with the following configurations and dependencies:

Spring Data JPA (https://spring.io/projects/spring-data-jpa): persistent data using the Java Persistence API

Express (http://expressjs.com/): minimalist web framework for NodeJs

No real decision process was involved in choosing a programming language or these frameworks. Using Java and Spring-Boot was clear from the beginning. The JPA was decided when deciding on the H2 database.
In addition we used the NodeJS environment to demonstrate the technology independence for microservices and as an easy to use environment to serve customized HTML.

Database

H2 Database (https://www.h2database.com/html/main.html): in memory database that supports JPA. Was modified to support permanent storage beyond runtime. The decision process is documented in design decision (“JPA Repository vs Database”) considered alternatives:
SQLite (https://www.sqlite.org/index.html),
MongoDB (https://www.mongodb.com/de)

2. System Requirements

In the following the systems requirements are discussed regarding the defined scope requirements in addition to internally set requirements as well as needed performance key figures for this type of architecture. In brackets you can find the team member responsible for the requirements realization.

2.1. Functional Requirements:

#1	Drop Down User Selection (David Cömert)
Description	Users should be able to log in to the system by providing correct credentials.
Priority	High
Impuls	User sends POST request via LogIn Form
Result	As Users are not verified via token or similar but simply by id, it is checked if the User exists in DB. User is presented with the appropriate functions.
Verification	Test cases

#2	Mark product (Lukas Sorer)
Description	User should be able to mark products for price changes
Priority	High
Impuls	User marks item in interface
Result	User receives an email if price of a marked item is reduced
Verification	Test cases

#3	Rate product (Dorina Karameti)
Description	Users should be able to rate purchased products
Priority	High
Impuls	User rates purchased product in interface form

Result	Rating is published and attached to respective item
Verification	Test cases

#4	Buy product (David Cömert)
Description	Users should be able to purchase any of the products available for purchase
Priority	High
Impuls	User sends GET Request via Interface to conduct a checkout
Result	User is forwarded to Success Window after checkout
Verification	Test cases/Manual Inspection

#5	Add item (Tobias Fühles)
Description	Vendors should be able to add new items to their inventories
Priority	High
Impuls	User sends POST Request via Interface to add item to DB.
Result	User receives JSON of added item
Verification	Test cases

#6	Remove item (Tobias Fühles)
Description	Vendors should be able to remove items from their inventories.
Priority	High
Impuls	User sends POST Request via Interface to delete item from DB
Result	User receives SUCCESS or FAILURE message
Verification	Test cases

#7	Adjust price (Lukas Sorer)
Description	Vendors should be able to change the price of their products
Priority	High
Impuls	User sends POST Request via Interface to change price of item in DB.
Result	User receives JSON of changed item
Verification	Test cases

#8	Adjust quantity (Tobias Fühles)
Description	Vendors should be able to change the quantity of their products
Priority	High
Impuls	User sends POST Request via Interface to change price of item in DB.
Result	User receives JSON of changed item
Verification	Test cases

#9	Calculate profits (Tobias Fühles)
Description	Vendors should be able to calculate the profit from the products they have sold
Priority	High
Impuls	User sends GET Request via Interface to get profit for sold items.
Result	User receives JSON with a single double value of profit by transaction history.
Verification	Test cases

#10	Track shipment (Kevin Miller)
Description	User should be able to track shipment of their sold products
Priority	High

Impuls	Users mark interest in shipment tracking.
Result	User receives email if shipment status is updated
Verification	Test cases

#11	Track history (Kevin Miller)
Description	Users and vendors should be able to track history of their transactions
Priority	High
Impuls	User selects GET request for transaction history view
Result	User is forwarded to transaction history view
Verification	Test cases

#12	Email notifications (Dorina Karameti)
Description	Users should receive emails about their marked products and shipment status
Priority	High
Impuls	See #2 #10
Result	User receives email(s)
Verification	Test cases

2.2. Non-functional requirements

#13	Performance (Kevin Miller)
Description	The system should execute User requests in a reasonable amount of time (e.g. < 500ms per User Request)
Priority	Medium
Impuls	Any user request

Result	Response time condition fulfilled
Verification	Manual Inspection

#14	Efficiency (Kevin Miller)
Description	The system endpoints should only explicitly executed what it is intended to do without triggering site effects or forwarding unnecessary data
Priority	High
Impuls	/
Result	/
Verification	Manual Inspection

#15	Access restriction (David Cömert)
Description	Customers should not have access to vendor services (e.g. add item to inventory), and customers and vendors should not have access to transaction and shipment information of other users
Priority	Medium
Impuls	/
Result	Functionalities should be adjusted dependent on User LogIn
Verification	Manual Inspection

#16	Data integrity (Dorina Karameti)
Description	Users should not have access to personal information of other users
Priority	Medium
Impuls	/
Result	Secure endpoints and information accessible in GUI

Verification	Manual Inspection

#17	Browser compatibility (Lukas Sorer)
Description	The interface shall run in the current versions of the common browsers (Firefox, Chrome)
Priority	High
Impuls	URI of the interface is accessed via browser.
Result	Interface runs
Verification	Manual Inspection

#18	Performance profit calculation (Tobias Fühles)
Description	The profit of an vendor should be calculated <4s after request
Priority	High
Impuls	User sends GET request via Interface
Result	Calculated profit is returned in time
Verification	Manual Inspection

3. 4+1 Views Model

3.1. Scenarios / Use Case View

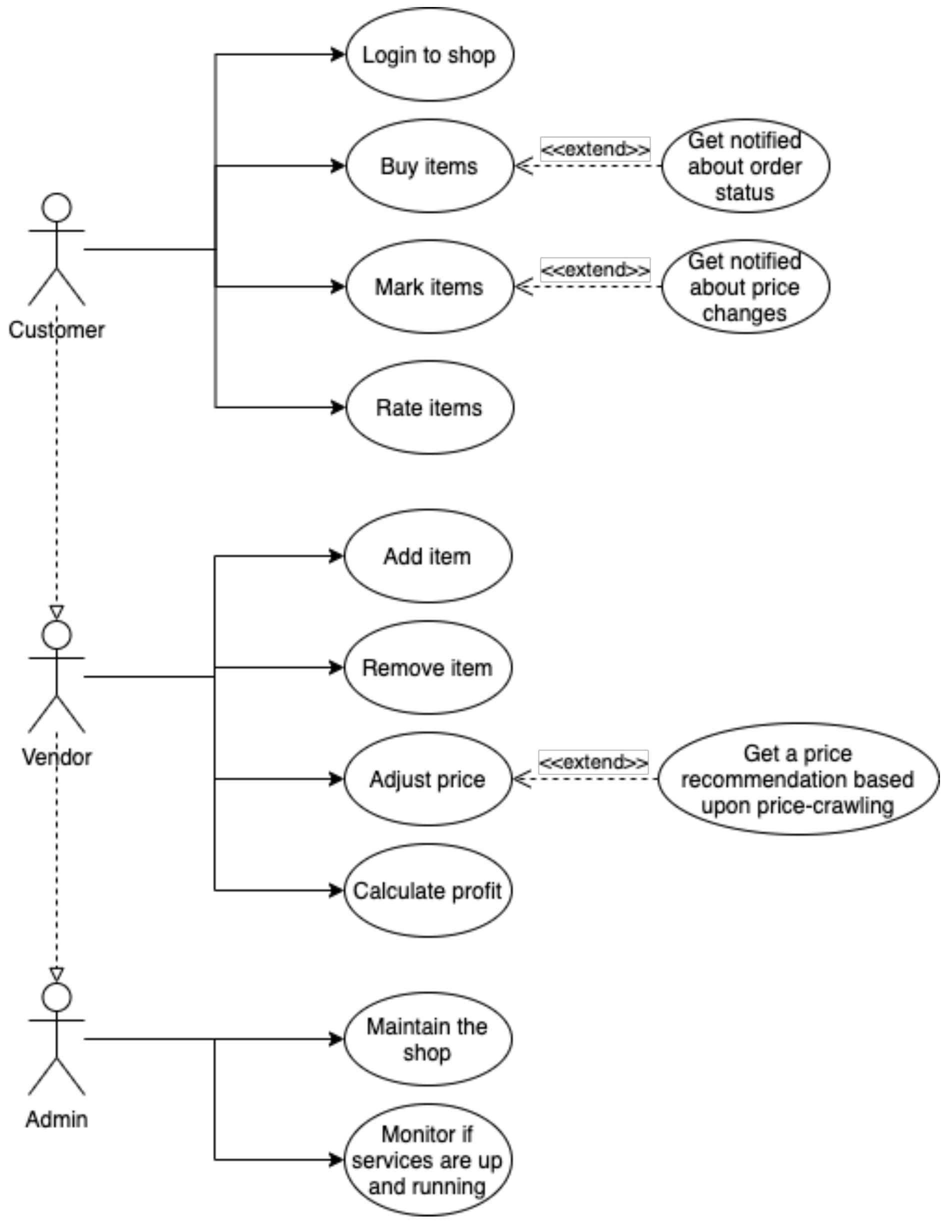
3.1.1. Use Case Diagram

The diagram below shows our use-cases on a high level of abstraction. It not only provides the reader with the three main roles of our project (i.e. the customer, vendor and admin) it also shows the functionality that can be accessed by each role.

Let’s start from the top with the customer and this person's possibilities to use our eShop. Although there is no explicit login functionality as one might know it from everyday applications such as an email client, the customer has to identify him- or herself by providing the system with a userId. The whole login topic was already discussed in chapter 1.1.2, to be more precise “Decision 8 - Login”, that’s why we won’t elaborate on this here any further. The next use-case that a user of type customer is able to do in our eShop is most certainly the foremost essential part of any ecommerce system: Buy items. This use-case is extended by a notification service that informs the user in a timely manner about the shipment status of his or her order. Furthermore there is the option to mark certain items of the webshop, to revisit them later on or, and here comes the notification service once again into play, to get notified when the price of the item changes. The last use-case that is enabled for the customer is the possibility to rate items on a scale from 1 - 5.

Next up is the user role vendor. This user class inherits all use-cases that are possible for the customer and has additional capabilities to interact with the eShop. For one the vendor can add items to the web shop. These are exactly the items that the customer then is able to buy. Moreover, as market conditions can change and bottlenecks in supply chains might occur, is it also possible for vendors to remove items from the webshop. Another use-case for the vendor is the adjustment of the price of a given item. This functionality is extended by a price-crawler that crawls through the internet, most likely another webshop, and proposes a price that was found examining how much the product costs at another webshop. The vendor of course has the option to fully ignore the proposed price and define his or her own price for a given item, the price-crawler just recommends a price to the vendor for orientation. The final use-case for the vendor is the contingency to calculate his or her profits. There the transaction history is of great help as it documents all bought (remember that the vendor inherits the functionality from the customer and is also eligible to buy items) and sold items for any vendor.

The last user role as described in the semester project paper is the admin, which inherits all functionality from the customer as well as from the vendor. On top of that the admin can maintain the eShop and monitor if services are up and running. Under maintenance we understand the updating of the system and the installation of new versions of the code. Due to the fact that we will containerize every microservice on its own there is the possibility that one of these containers might experience a fallout which would lead to problems with the whole system (some more critical than others). The admin needs to keep an eye if all containers are up and running properly, this is what we understand as monitoring.



3.1.2. Use Case Descriptions

Below we have added the Use Case Descriptions. In these diagrams the different possible scenarios are explained in more detail. We included all main scenarios we believe are relevant to our shop.

Use Case:	**Mark a product**
Use Case ID:	1.1
Actor(s):	Customer
Brief Description:	The customer marks an item of interest from the shop for price changes
Pre-Conditions:	
Post-Conditions:	1. Item is saved as marked for specific user
Main Success Scenario:	1. Customer marks an item 2. Item is checked regularly for price changes 3. User is notified if/when price changes
Extensions:	
Priority:	High
Performance Target:	800 products marked a day
Issues:	-

Use Case:	Rate Product
Use Case ID:	1.2
Actor(s):	Customer
Brief Description:	The customer rates an item
Pre-Conditions:	The customer has purchased the item
Post-Conditions:	The rating of the item has to be updated
Main Success Scenario:	1. The customer locates to item rating interface 2. The customer rates the item 3. The customer submits the rating
Extensions:	
Priority:	Medium
Performance Target:	500 ratings a day
Issues:	-

Use Case:	Add to Cart
Use Case ID:	1.3
Actor(s):	Customer
Brief Description:	The customer adds an item from the shop to their cart

Pre-Conditions:	
Post-Conditions:	1. Cart is updated with new item
Main Success Scenario:	1. The customer opens item browsing interface 2. The customer adds item(s) to cart
Extensions:	
Priority:	High
Performance Target:	100 adds a day
Issues:	-

Use Case:	**Purchase Product(s)**
Use Case ID:	1.4
Actor(s):	Customer
Brief Description:	The customer buys an item from the shop
Pre-Conditions:	
Post-Conditions:	The order details of the user are saved
Main Success Scenario:	1. The customer locates to cart 2. The customer checks out 3. The customer enters shipping details
Extensions:	The history of the transaction is saved
Priority:	High

Performance Target:	200 purchases a day
Issues:	-

Use Case:	**Add item to shop**
Use Case ID:	2.1
Actor(s):	Vendor
Brief Description:	The vendor adds a product to their inventory
Pre-Conditions:	User needs correct role and permission to access inventory
Post-Conditions:	A price recommendation is returned
Main Success Scenario:	1. Vendor opens inventory management interface 2. Vendor adds product to their inventory 3. Vendor receives price recommendation 4. Vendor accepts/rejects price recommendation 5. Vendor sets/updates price 6. Vendor sets quantity
Extensions:	
Priority:	High
Performance Target:	200 items added a week
Issues:	-

Use Case:	Remove item from shop
Use Case ID:	2.2
Actor(s):	Vendor
Brief Description:	The vendor removes a product from their inventory
Pre-Conditions:	User needs correct role and permission to access inventory
Post-Conditions:	Item is no longer available for purchase
Main Success Scenario:	1. Vendor opens inventory management interface 2. Vendor removes product from their inventory
Extensions:	
Priority:	High
Performance Target:	200 items removed a week
Issues:	-

Use Case:	Adjust price
Use Case ID:	2.3
Actor(s):	Vendor
Brief Description:	The vendor changes the price of a product
Pre-Conditions:	1. User needs correct role and permission to access inventory 2. Item is available in inventory

Post-Conditions:	Price of product is updated
Main Success Scenario:	1. Vendor opens inventory management interface 2. Vendor enters new price for the item 3. Vendor saves new item
Extensions:	
Priority:	High
Performance Target:	200 items adjusted a week
Issues:	-

Use Case:	**Adjust quantity**
Use Case ID:	2.4
Actor(s):	Vendor
Brief Description:	The vendor sets the number of items available in stock
Pre-Conditions:	1. User needs correct role and permission to access inventory 2. Item is available in inventory
Post-Conditions:	Item quantity is updated
Main Success Scenario:	1. Vendor opens inventory management interface 2. Vendor enters new product quantity 3. Vendor saves new quantity
Extensions:	

Priority:	High
Performance Target:	200 items adjusted a week
Issues:	-

Use Case:	**Calculate profits**
Use Case ID:	2.5
Actor(s):	Vendor
Brief Description:	Calculate profit of items sold by a vendor
Pre-Conditions:	User needs correct role and permission to access inventory
Post-Conditions:	
Main Success Scenario:	1. Vendor opens inventory management interface 2. Vendor clicks on calculate profit 3. Profit is calculated 4. The calculated profit is returned
Extensions:	
Priority:	High
Performance Target:	200 times a day
Issues:	-

Use Case:	Track shipment
Use Case ID:	3.1
Actor(s):	Customer
Brief Description:	The customer tracks their shipment
Pre-Conditions:	1. The customer has ordered an item 2. The customer has provided an address
Post-Conditions:	
Main Success Scenario:	1. Customer opens track shipment interface 2. Customer is shown shipment status
Extensions:	The customer can also subscribe to receive emails when shipment status changes
Priority:	Medium
Performance Target:	50 items per customer
Issues:	-

Use Case:	Track history
Use Case ID:	3.2
Actor(s):	Customer, Vendor
Brief Description:	The customer, vendor track their history

Pre-Conditions:	1. A) The customer has ordered a product B) The vendor has item(s) in inventory
Post-Conditions:	
Main Success Scenario:	1. User opens track shipment interface 2. User is shown transaction history
Extensions:	
Priority:	Medium
Performance Target:	50 items per user
Issues:	-

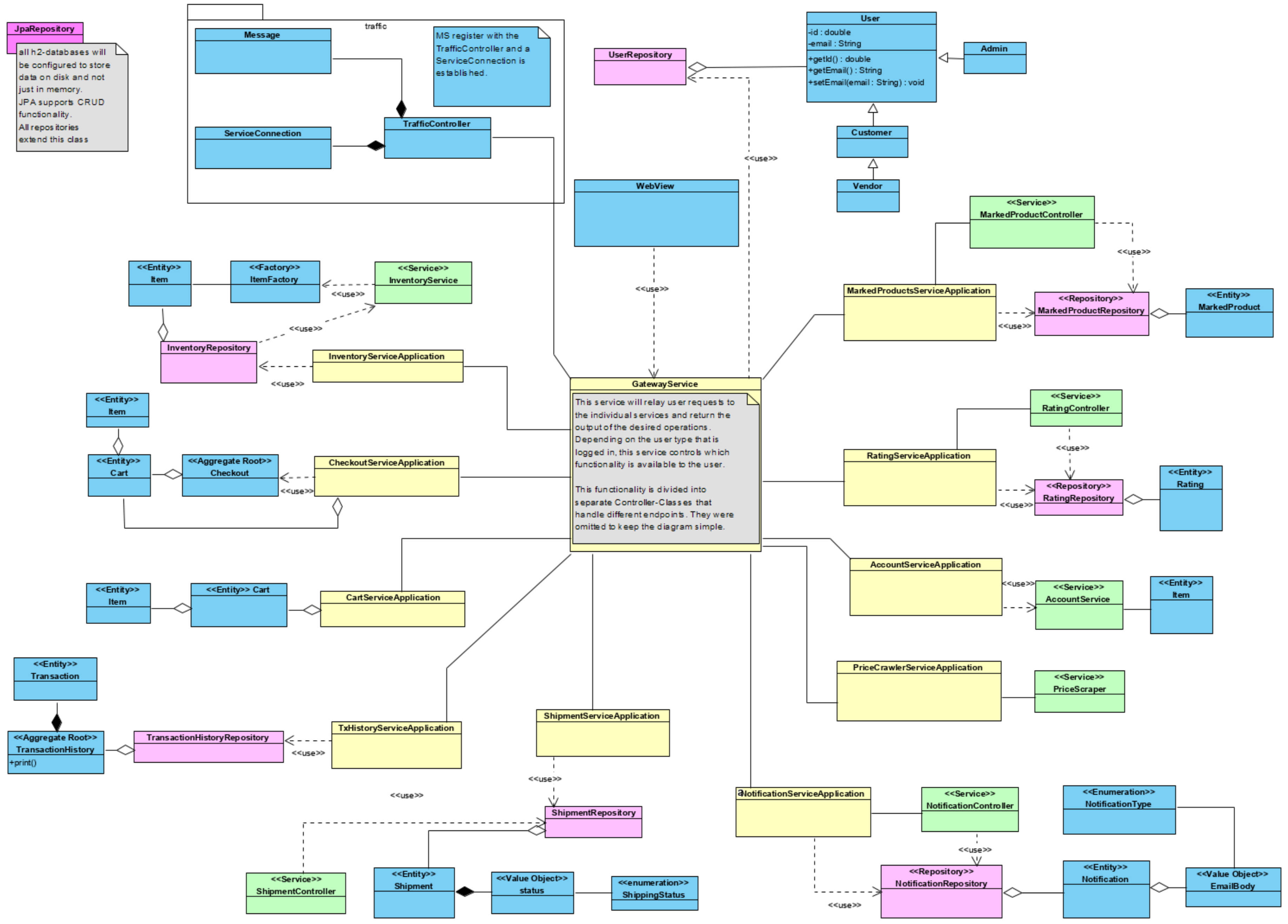
3.2. Logical View

Logical Overview

The following class diagram represents the Architectural Overview of 1.1.3 as a UML class diagram. As mentioned, all Microservices (yellow) are connected to the Gateway Service, and the only way for the Gateway Service to interact with any entities that make up the identity of the shop is through these microservices.

The Gateway Service also contains a TrafficController which is responsible for enabling microservices to register with the system in order to support the required publish/subscribe pattern.

Afterwards you will find class diagrams of every individual service. Please note, that these are supplement the “big picture” and slot in where you would find the respective service in the architectural overview.


JpaRepository
traffic
Message
ServiceConnection
TrafficController
User
-id : double
-email : String
+getId() : double
+getEmail() : String
+setEmail(email : String) : void
Admin
Customer
Vendor
UserRepository
<<use>>
WebView
<<Service>> MarkedProductController
MarkedProductsServiceApplication
<<Repository>> MarkedProductRepository
<<Entity>> MarkedProduct
<<Entity>> Item
<<Factory>> ItemFactory
<<Service>> InventoryService
InventoryRepository
InventoryServiceApplication
GatewayService
<<Entity>> Item
<<Entity>> Cart
<<Aggregate Root>> Checkout
CheckoutServiceApplication
<<Service>> RatingController
RatingServiceApplication
<<Repository>> RatingRepository
<<Entity>> Rating
<<Entity>> Item
<<Entity>> Cart
CartServiceApplication
AccountServiceApplication
<<Service>> AccountService
<<Entity>> Item
PriceCrawlerServiceApplication
<<Service>> PriceScraper
<<Entity>> Transaction
<<Aggregate Root>> TransactionHistory
+print()
TransactionHistoryRepository
TxHistoryServiceApplication
ShipmentServiceApplication
ShipmentRepository
<<Service>> ShipmentController
<<Entity>> Shipment
<<Value Object>> status
<<enumeration>> ShippingStatus
NotificationServiceApplication
<<Service>> NotificationController
<<Enumeration>> NotificationType
<<Repository>> NotificationRepository
<<Entity>> Notification
<<Value Object>> EmailBody

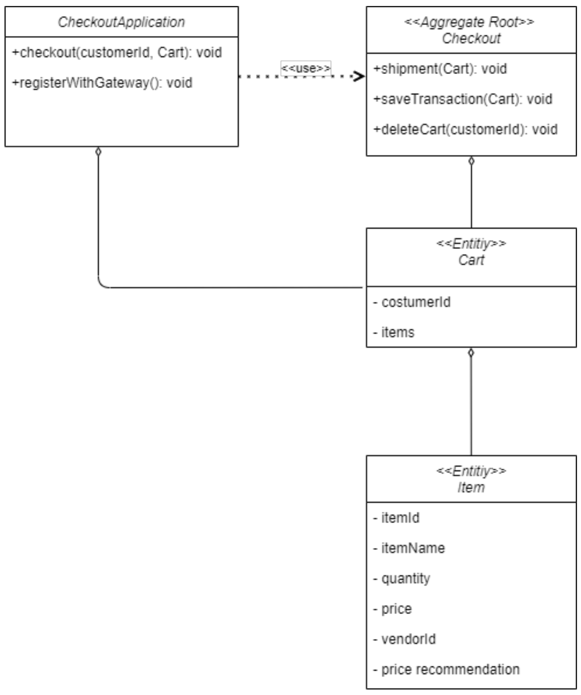
all h2-databases will be configured to store data on disk and not just in memory.
JPA supports CRUD functionality.
All repositories extend this class

MS register with the TrafficController and a ServiceConnection is established.

This service will relay user requests to the individual services and return the output of the desired operations. Depending on the user type that is logged in, this service controls which functionality is available to the user.

This functionality is divided into separate Controller-Classes that handle different endpoints. They were omitted to keep the diagram simple.

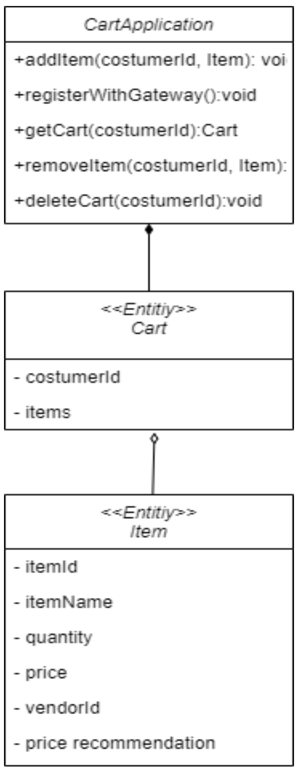
Checkout Service



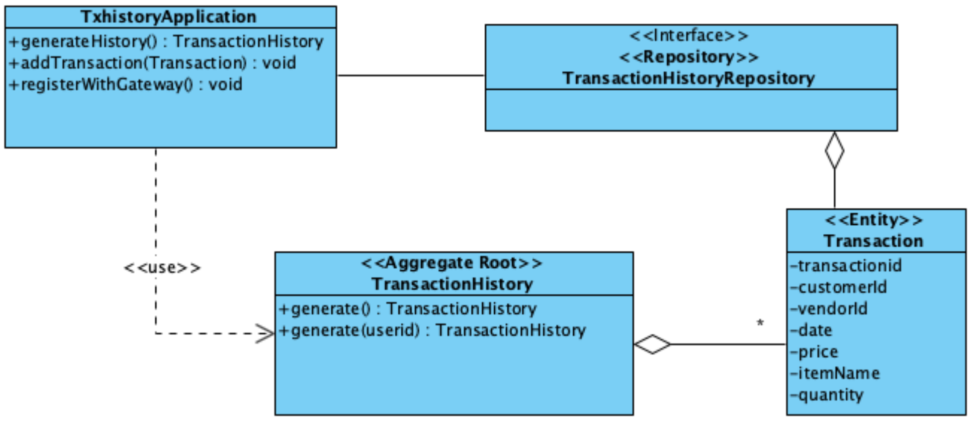
The checkout service is in charge of distributing the items in the cart to shipment and to save the transaction. In addition, if all works well, the cart should be deleted afterwards. Shipment, saveTransaction and deleteCart send a JSON via http to the gateway.

Cart Service

The cart service manages each cart per customer. The cart is reachable after he registers on the gateway. Only one cart per endpoint can be changed.

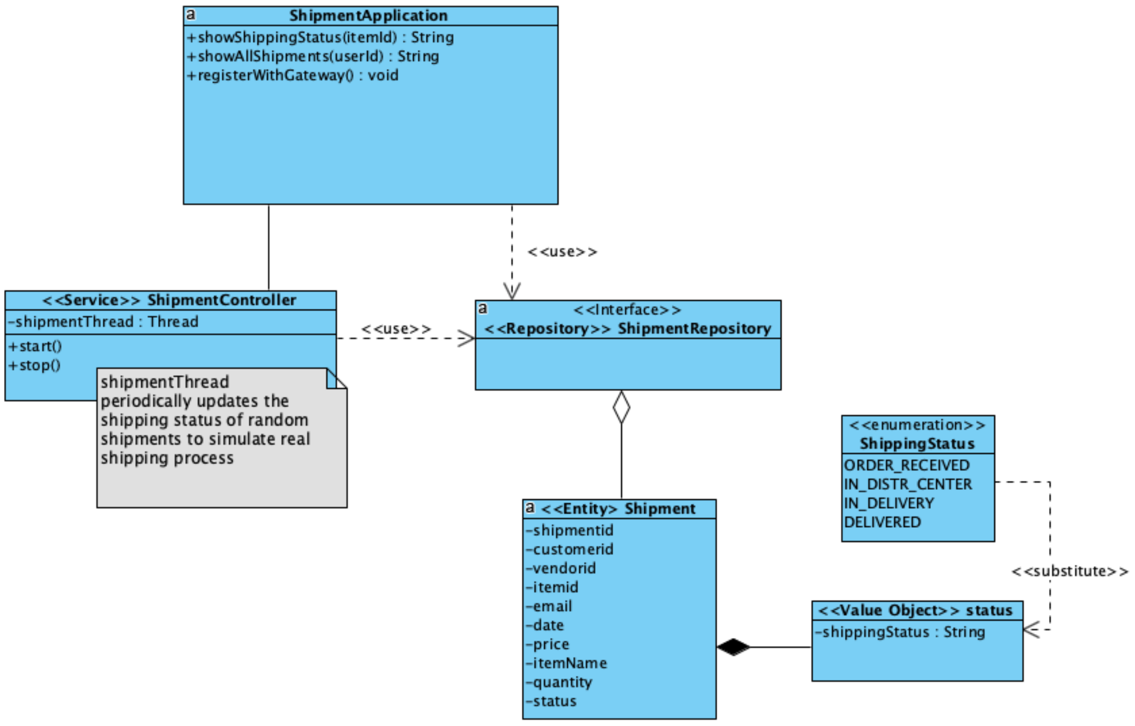

CartApplication
+addItem(costumerId, Item): voi
+registerWithGateway():void
+getCart(costumerId):Cart
+removeItem(costumerId, Item):
+deleteCart(costumerId):void
<<Entitiy>>
Cart
- costumerId
- items
<<Entitiy>>
Item
- itemId
- itemName
- quantity
- price
- vendorId
- price recommendation

Transaction History Service (TxHistory)


TxhistoryApplication
+generateHistory() : TransactionHistory
+addTransaction(Transaction) : void
+registerWithGateway() : void
<<Interface>>
<<Repository>>
TransactionHistoryRepository
<<use>>
<<Aggregate Root>>
TransactionHistory
+generate() : TransactionHistory
+generate(userid) : TransactionHistory
*
<<Entity>>
Transaction
-transactionid
-customerId
-vendorId
-date
-price
-itemName
-quantity

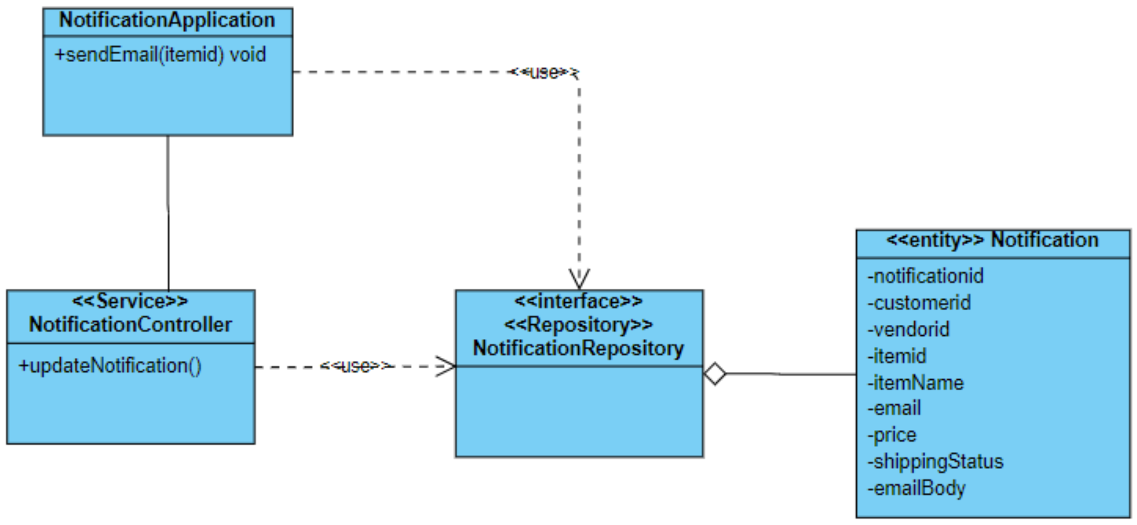
The transaction history service is used to store all transactions that occur within the shop ecosystem. All endpoints are implemented in the TxhistoryApplication class. The TransactionHistoryRepository stores all Transaction entities. The TransactionHistory is an aggregation of an unlimited number of Transactions.

Shipment Service



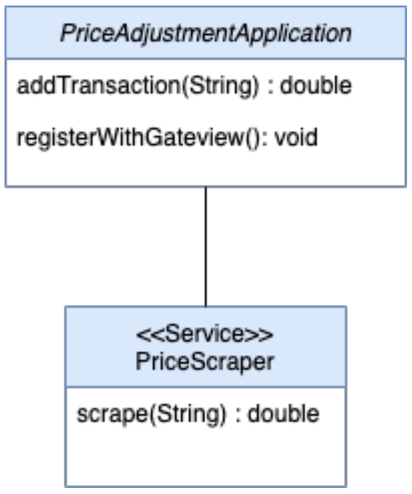
The Shipment Service simulates the shipping process of bought items. All Shipment entities are stored in the ShipmentRepository. The ShipmentController contains a thread which updates the status of Shipments randomly when started. When a new Shipment is created, its initial status is set to ORDER_RECEIVED. It can be stopped for maintenance and has to be started manually.

Notification Service



The notification service is used to send email notifications to the user in one of two cases: (1) If they have marked an item via the marked product service, they will receive an email in case of a price change and (2) if they have purchased a product from the shop they will be notified about their shipment status when there is an update. The service consists of a repository that stores all the information about the notifications and a notification application that communicates with the gateway.

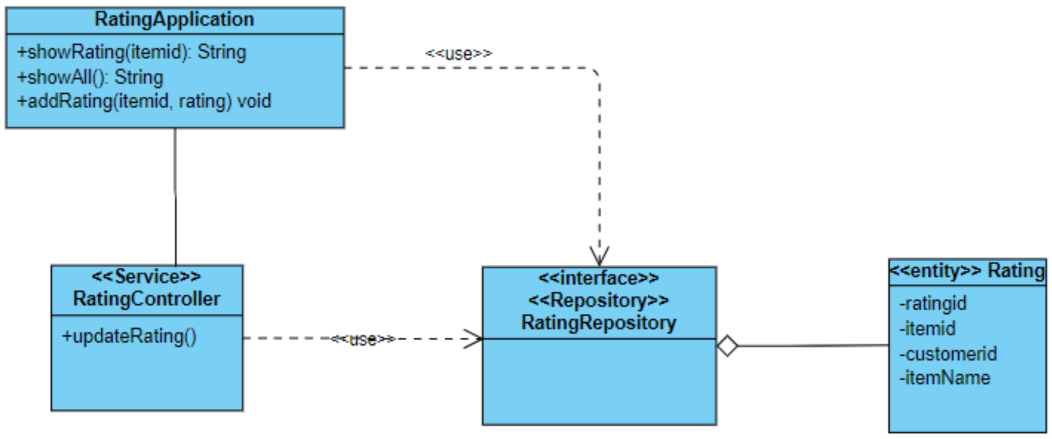
Price Crawler Service (Price adjustment)



PriceAdjustmentApplication
addTransaction(String) : double
registerWithGateview(): void
<<Service>> PriceScraper
scrape(String) : double

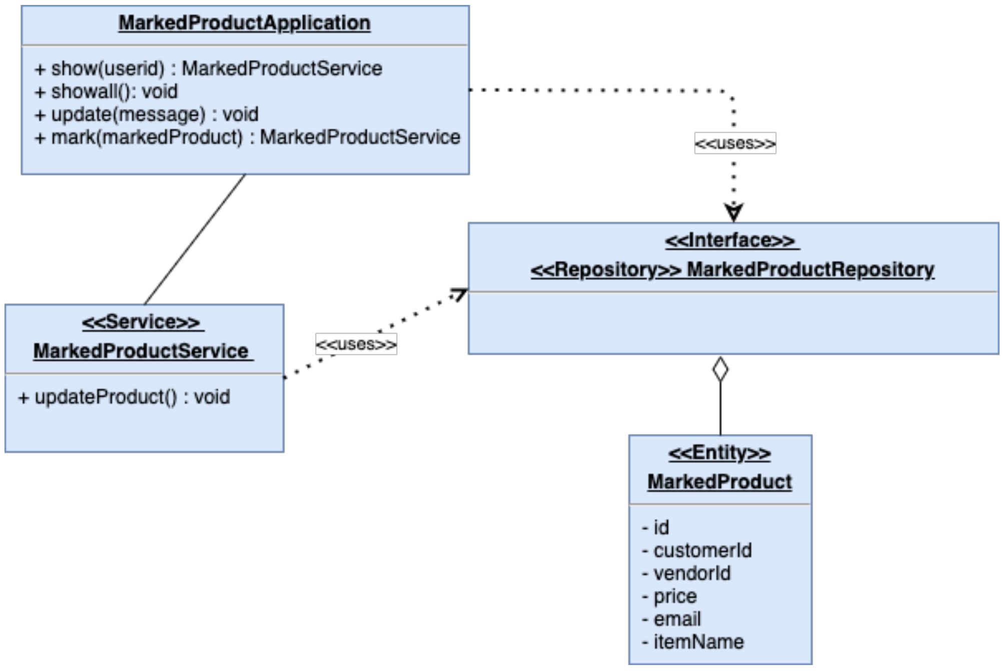
The price crawler service (or sometimes "price adjustment service") consists of a price adjustment application part and a price scraper service part. The application part is responsible for establishing a connection to the gateway service and the streamline communication between them. The price scraper service is responsible for crawling the internet, given a product name, and returning a recommended price that was retrieved from another webshop. In contrast to almost every other class it does not contain a repository or an entity as it just receives a product name, crawls the web for this value and returns a price if found. No permanent storage needed.

Rating Service



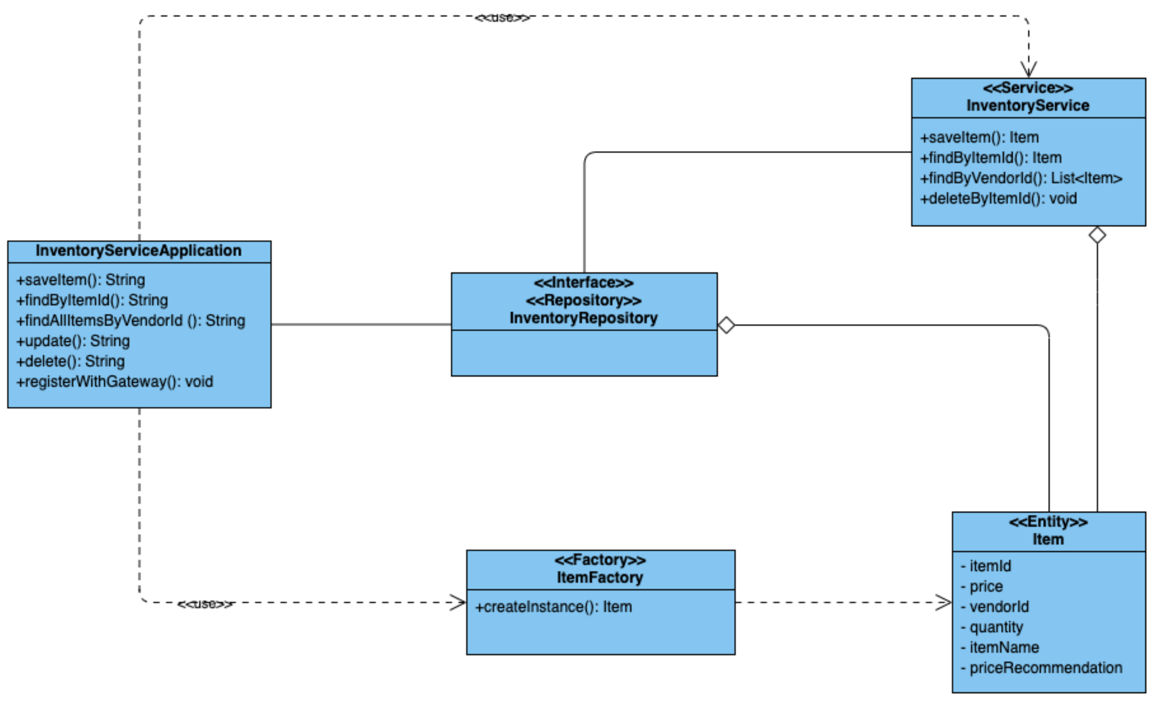
The Rating Service is responsible for storing and calculating the ratings of products. It consists of a RatingRepository where all the ratings are stored and a RatingApplication which calculates with ratings or products and communicates with the gateway.

Marked Product Service



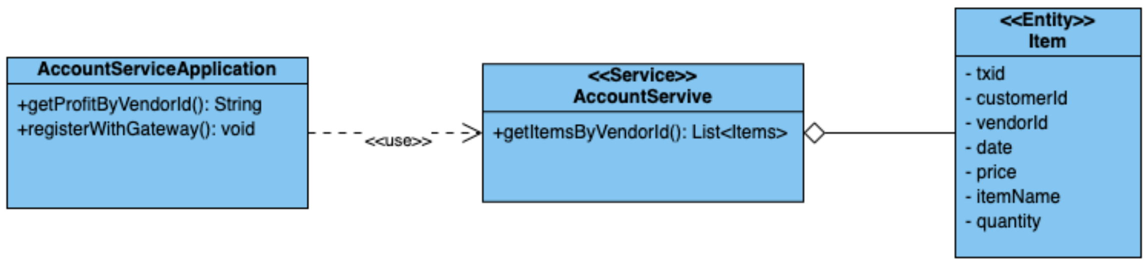
The marked product service is used to allow the user to mark items of his or her choice. It consists of the marked product application that talks to the gateway, a marked product service that realises price changes and triggers the notification service to inform customers about changes in price, a repository that stores all information about the marked items.

Inventory Service



The inventory service is responsible for providing the CRUD functionality for all items within the eshop ecosystem as well as additional services like find all Items for a specific vendor. Respective endpoints can be found in the InventoryServiceApplication as a Controller class only was used before gateway integration. Item Factory is used as a pattern to be able to manipulate the creation process of an Item.

Account Service



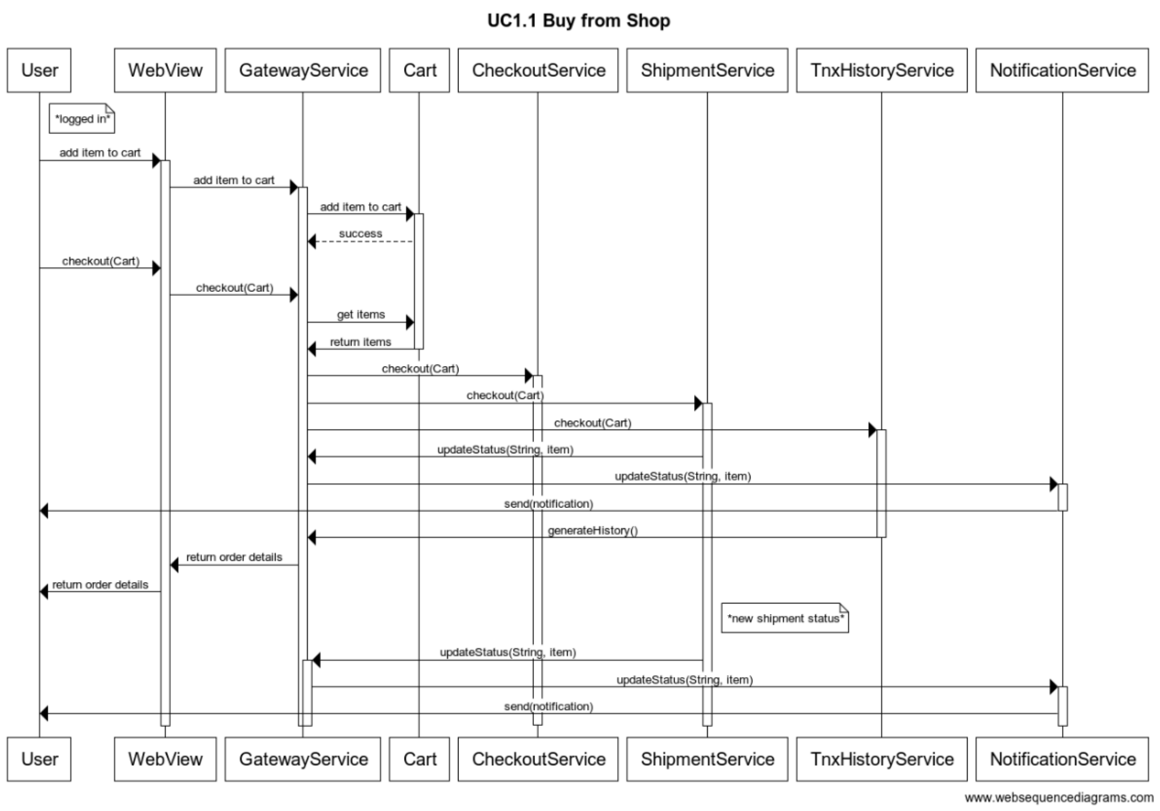
AccountServiceApplication
+getProfitByVendorId(): String
+registerWithGateway(): void
<<use>>
<<Service>>
AccountServive
+getItemsByVendorId(): List<Items>
<<Entity>>
Item
- txid
- customerId
- vendorId
- date
- price
- itemName
- quantity

The account service is responsible for the profit (at this stage just the revenue) calculation. To achieve this functionality a transaction history endpoint is fetched which returns a list of Items used to sum up the profit. As no request is stored this service comes without a repository.

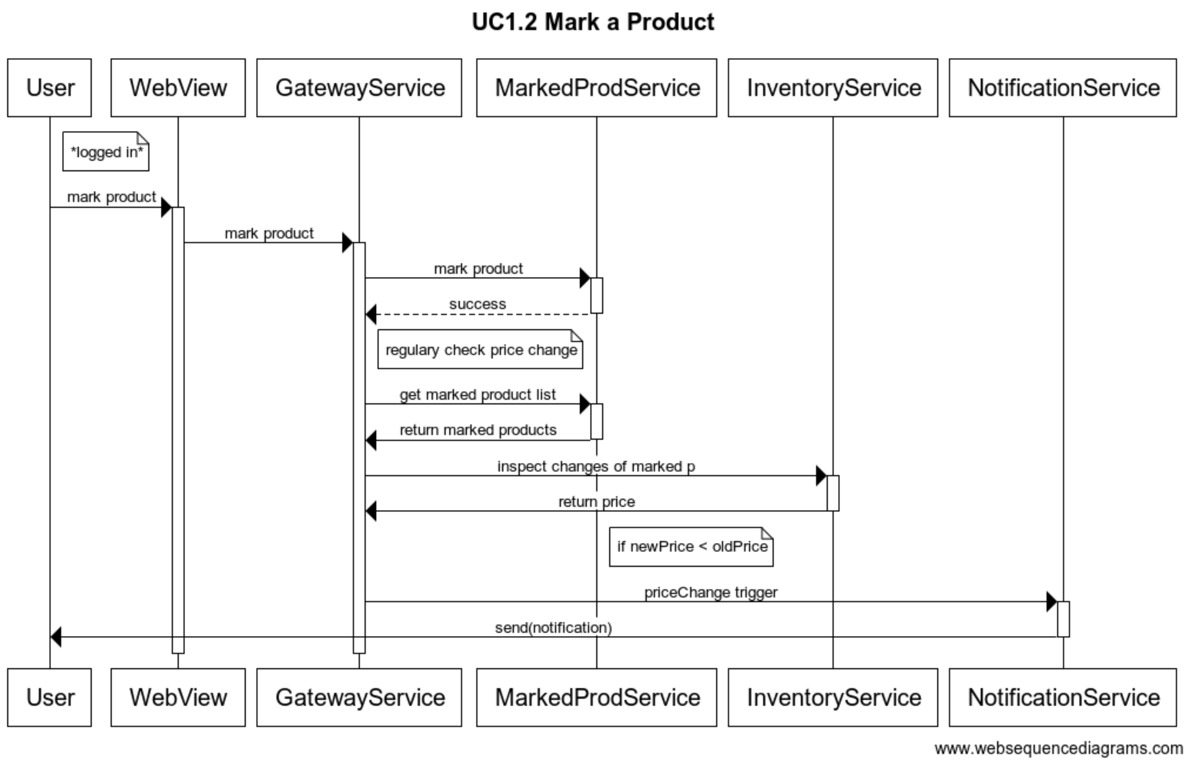
3.3. Process View

In the following we will take an extensive look into the processes that are involved while using the webshop. We chose the notation of sequence diagrams in order to visualise the proceedings of the background as well of what's visible to the actual user. The namespaces and enumeration is based upon the tabular use-case description as seen in chapter “3.1.2 Use Case Description”. We won’t get into that much detail as the diagrams encapsulate a high density of information. The textual description will just elaborate on the thought-process and discussions we had during designing these. Upfront, and this is already mentioned in our design decisions (to be more precise decision 2 - The Gateway) the most severe change to the first version of the process view came from the decision to either let services communicate between each other or to consolidate and allow communication only to happen through the gateway. Although there are some positive aspects if services communicate between each other (the communication sequence might be easier) we decided against this as it would tremendously increase the potential for errors in runtime when e.g. services fall out. With one central point of communication, that additionally inspects if services are up and running, we hoped to ensure a better debugging experience as well as enabling the overall system to be more adaptable. This comes from the fact that if we would have chosen the architecture in which services talk to each other, the amount of changes would dramatically increase as new modules and services would be added to the current version. By streamlining all communication through the gateway service we would only have to change the code or add new controller classes in the gateway and would not have to care about other dependencies within the service-to-service interaction.

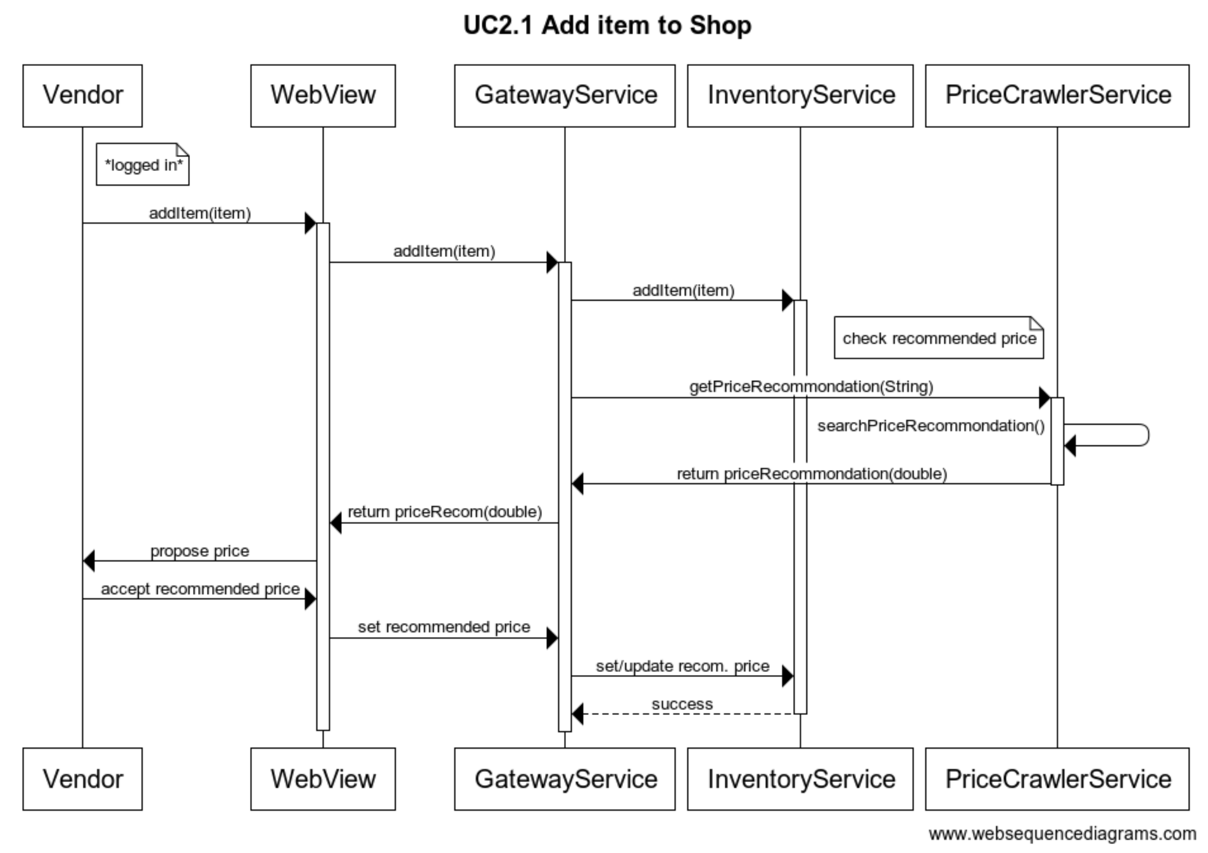
The first sequence diagram, and probably the most important for any user of the eShop ist the “UC1.1 Buy from Shop”-diagram. It shows how the different services talk to the gateway service, how the gateway then returns results and important information back to the web-view (the front end) in order to perform an order-to-cash process (without taking financial transactions into account). We see that the process includes many core services, to name a few; cart, checkout service, shipment service and transaction history service, that are essential to conduct this process. We see that the purchase is realised when the transaction history service forwards the successful logging of an purchase to the gateway which then forwards this once again to the web-view so that the user can see it. As the order travels along to its destination the shipment service informs the gateway that the status of the order has changed. Once again the gateway passes this information to the notification service that then sends a notification to the customer containing the new status of the shipment.



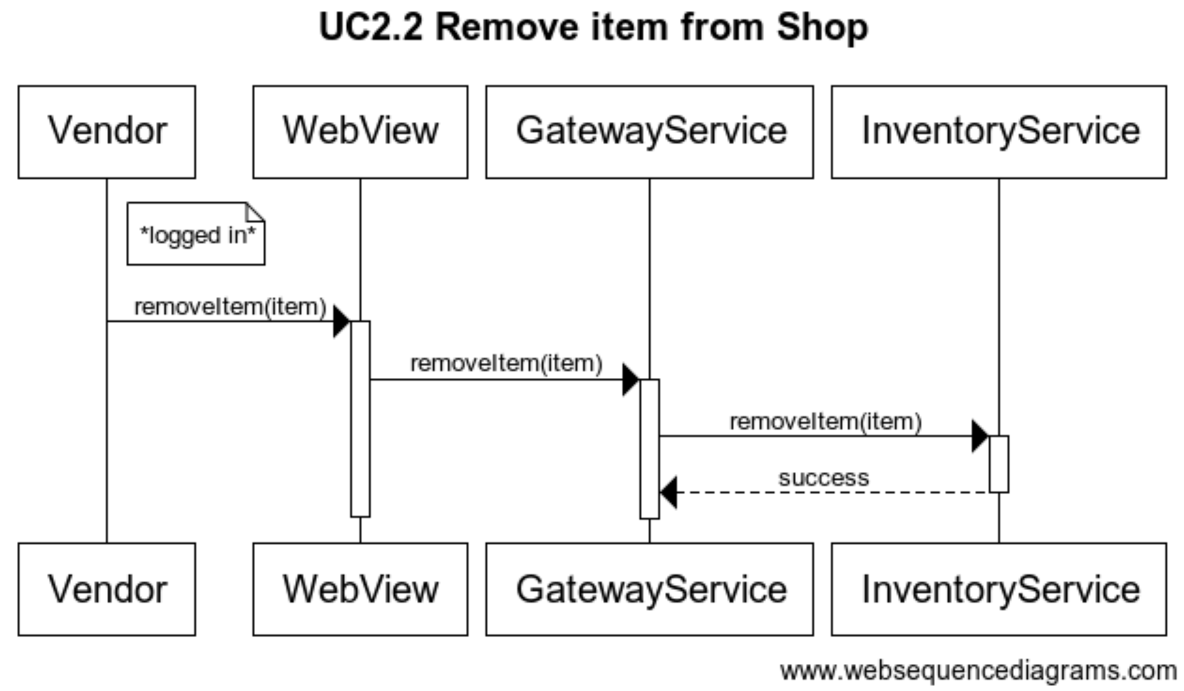
The next sequence diagram “UC1.2 Mark a Product” as shown below, describes the proceedings that happen so that a product is marked in the webshop. We assume that mainly the customers will use this possibility. As already seen in the first sequence diagram and elaborated in our design decisions all communication goes through the gateway. We presume that a user is logged into the webshop and has found a product that he or she likes, but where the price is too high for the customer. The customer then has the possibility to mark one or multiple items and will get notified if the price decreases on these items.



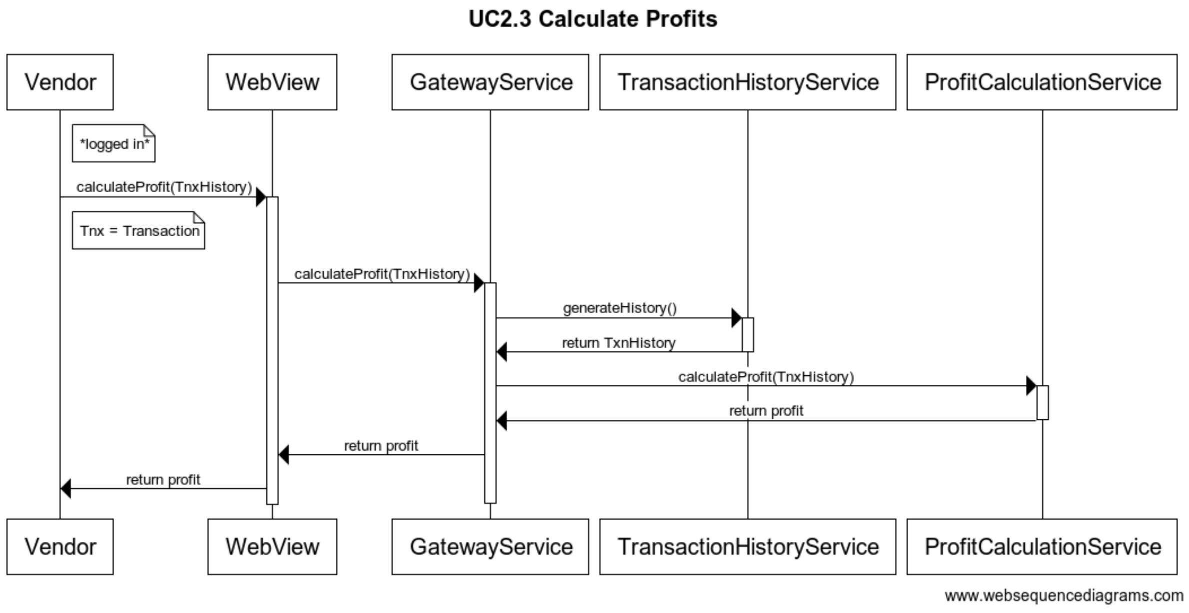
Next up is the sequence diagram “UC1.3 Rate a Product”. This feature helps the customers to identify products that evoked high customer satisfaction and to omit products that were a disappointment to the person ordering them. This sequence diagram is the last one of the customer-centric diagrams, where the customer, a potential buyer of the eShop, is the main focus of consideration.



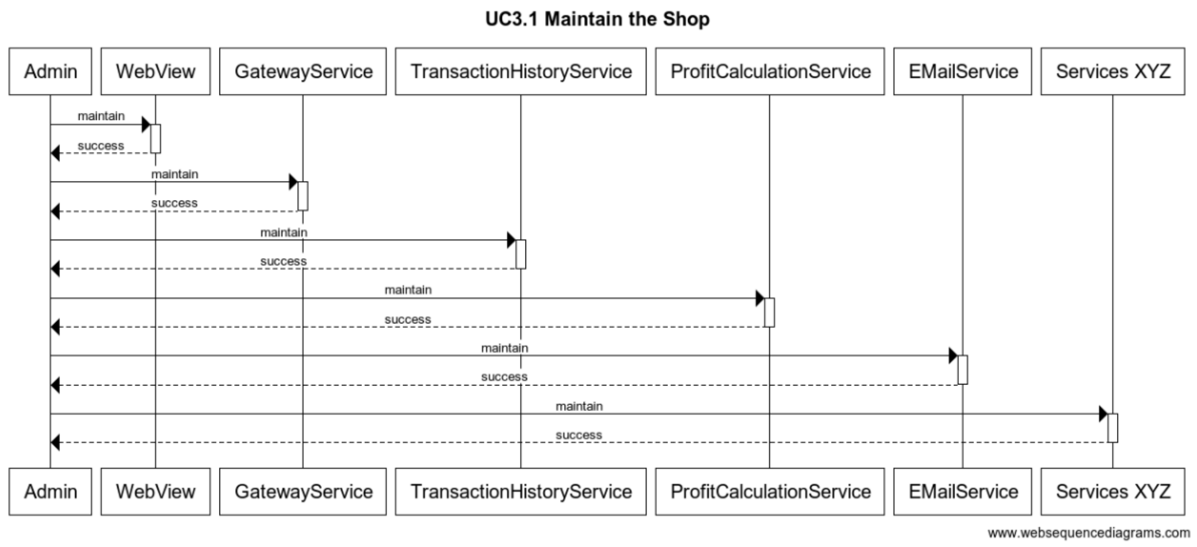
The first sequence diagram at which the vendor is the main focus of interest is “UC2.1 Add item to Shop”. Throughout this document we use the words “item” and “product” synonymously for an object that can be bought at the webshop. Once more we can see that the gateway plays an important role in streamlining communication between the different services. We integrated the feature of the price crawler into this diagram in order to showcase how this vendor-sided extra helps the vendor to find the perfect price for his or her product.



"UC2.2 Remove items from Shop" is pretty straight forward the sequence diagram that shows the steps taken by the system in order to remove an item that was formally added to the webshop by a vendor, from the shop.

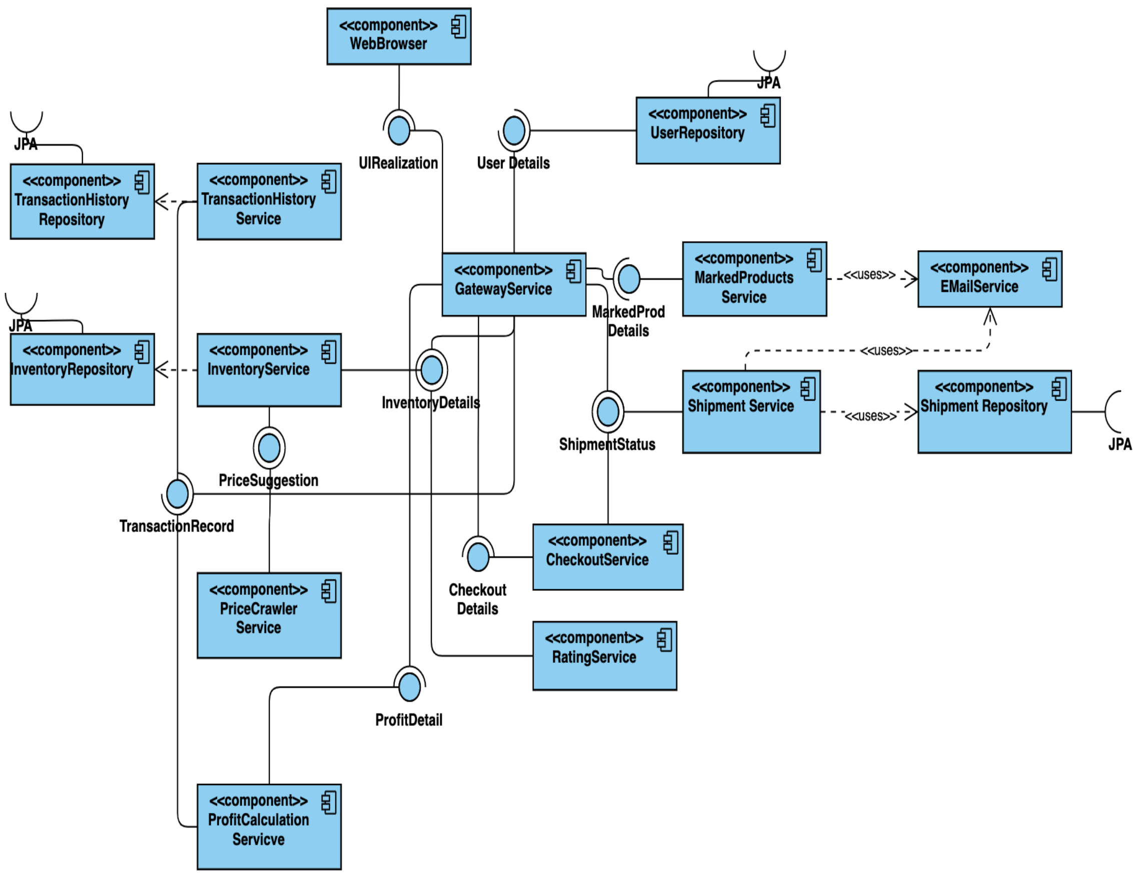


The sequence diagram "UC2.3 Calculate Profits" is the last diagram of the vendor-focused series of sequence diagrams. It describes how the transaction history service returns, by asking the gateway to forward the request, to the profit calculation service the logged information about how many products were sold and at which price. It aggregates this information and returns a sum to the vendor that asked for this information.



Finally we can talk about the sequence diagram “UC3.1 Maintain the Shop” which is focused on the administrators side of things. As we understand the administrator as the person that is responsible for the undisturbed functioning of the webshop, he or she will have access to the source code in order to fix trivialities that most certainly will occur during runtime. We decided to model this diagram in a highly abstract way, as we cannot predict which services and aspects might make problems.

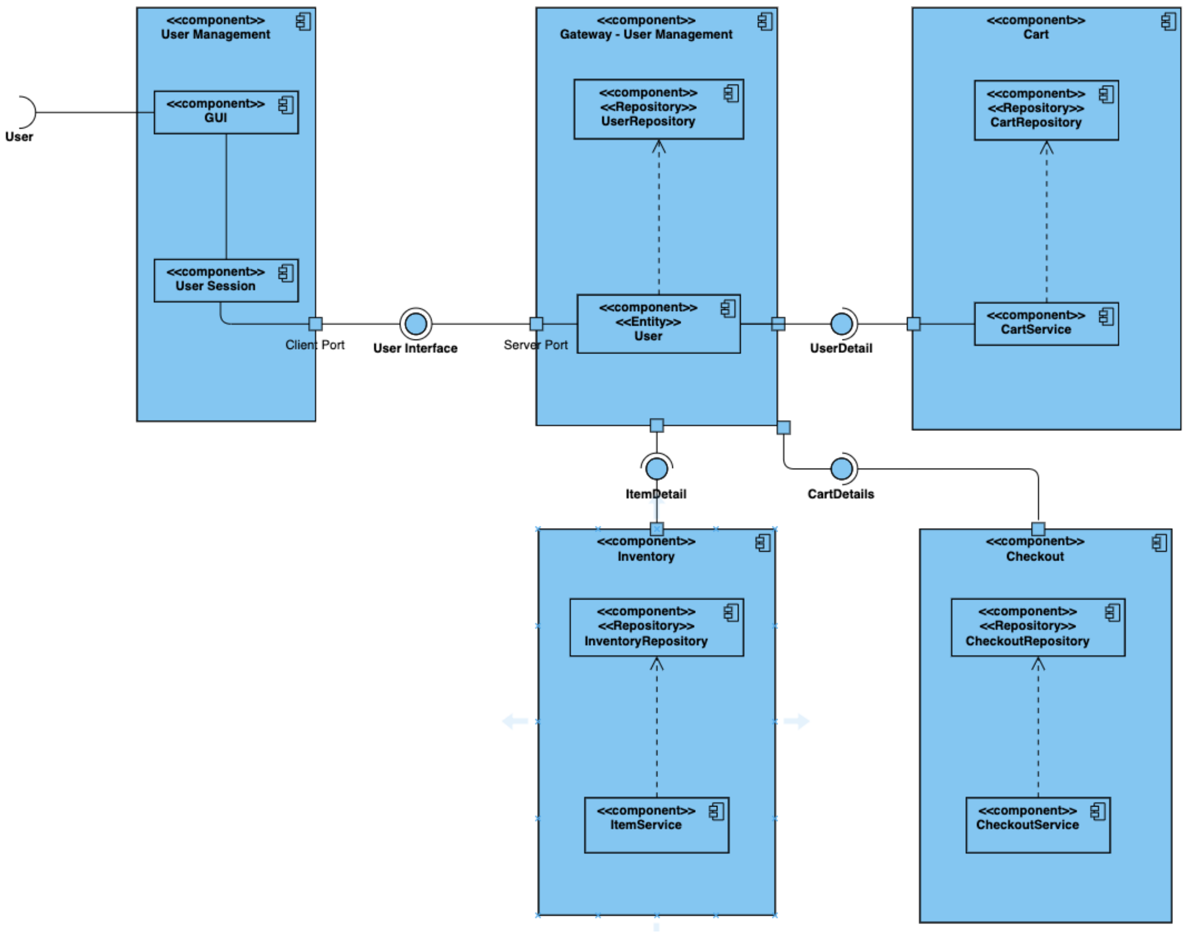
3.4. Development View



<<component>> WebBrowser
<<component>> UserRepository
JPA
UIRealization
User Details
JPA
<<component>> TransactionHistory Repository
<<component>> TransactionHistory Service
<<component>> GatewayService
<<component>> MarkedProducts Service
<<uses>>
<<component>> EMailService
MarkedProd Details
JPA
<<component>> InventoryRepository
<<component>> InventoryService
InventoryDetails
<<uses>>
<<component>> Shipment Service
<<uses>>
<<component>> Shipment Repository
JPA
ShipmentStatus
PriceSuggestion
TransactionRecord
<<component>> CheckoutService
Checkout Details
<<component>> PriceCrawler Service
<<component>> RatingService
ProfitDetail
<<component>> ProfitCalculation Servicve

This view shows how the main components of the Microservices are wired together with a representational focus on service components, representing the individual functionalities/classes as executables, and the respective used repositories. As entities, controllers and other used patterns are shown in the class diagrams they are not included in this view.

Most services are connected with their port to the gateway, but occasionally also communicating with each other. By now we are not entirely sure whether the communication will only be realized via the gateway - meaning micro services depending on information from each other would use the gateway as middleman. This decision will be made for the FINAL submission.



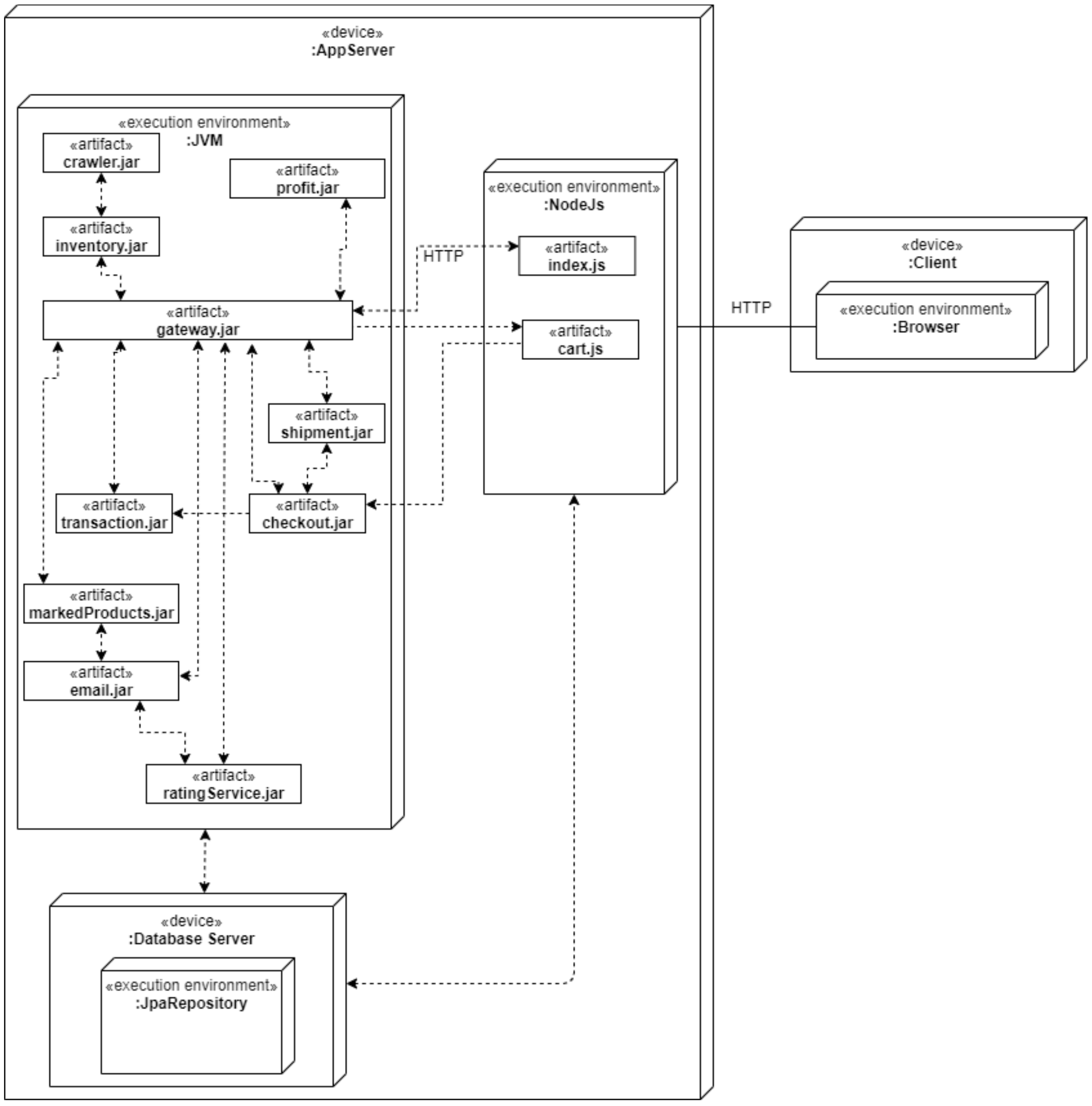
<<component>> User Management
<<component>> GUI
<<component>> User Session
User
Client Port
User Interface
Server Port
<<component>> Gateway - User Management
<<component>> <<Repository>> UserRepository
<<component>> <<Entity>> User
UserDetail
<<component>> Cart
<<component>> <<Repository>> CartRepository
<<component>> CartService
ItemDetail
CartDetails
<<component>> Inventory
<<component>> <<Repository>> InventoryRepository
<<component>> ItemService
<<component>> Checkout
<<component>> <<Repository>> CheckoutRepository
<<component>> CheckoutService

Figure: Overview of Component Diagram relevant for “OrderItem as Customer”

This figure shows a simplified structure with relevant components/interfaces needed to select an Item as a User followed by a checkout.

3.5. Physical View

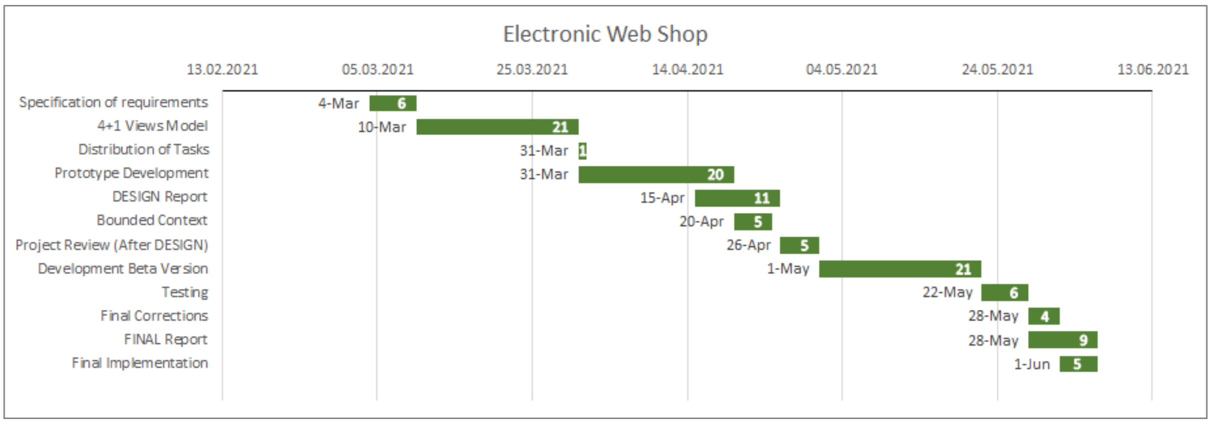
The physical view shows the deployment of the application. The application runs on one machine (AppServer), but can easily be deployed on different systems. This is possible with the http connections between the micro services.



4. Team Contribution and Continuous development method

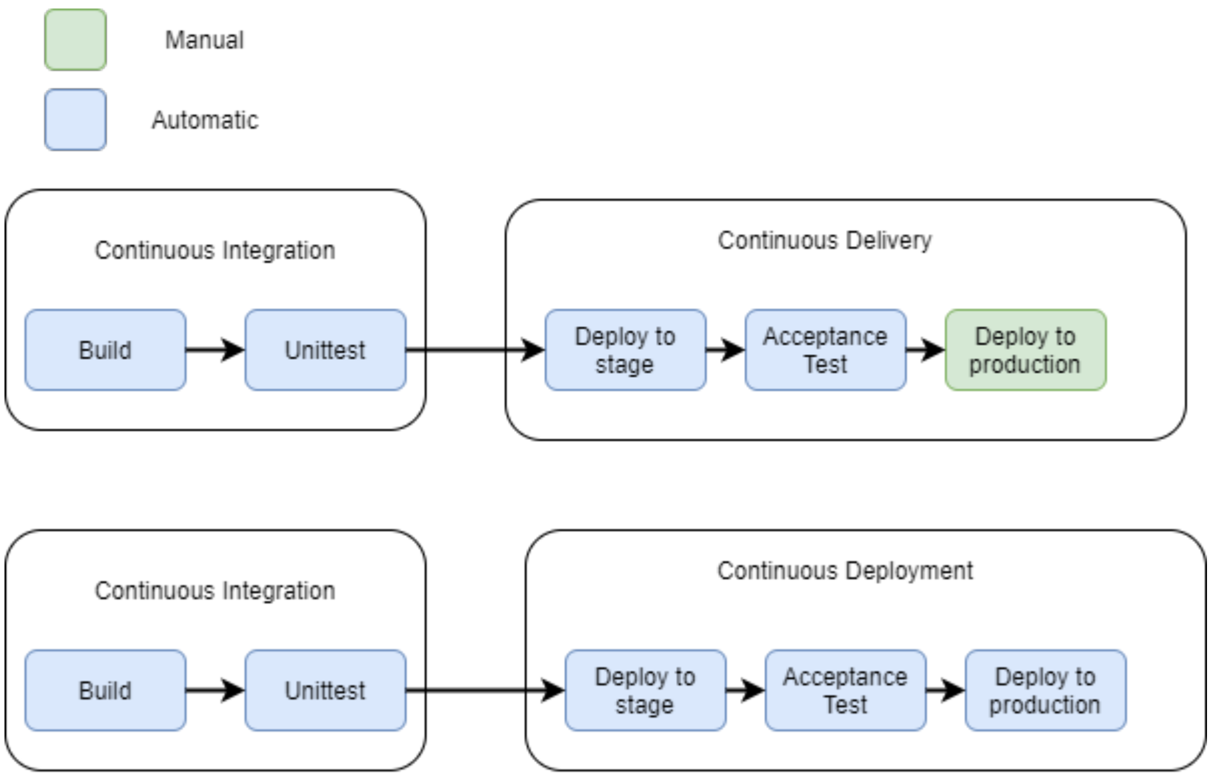
4.1. Project Tasks and Schedule

To better visualize our tasks and schedule we created a Gantt chart in accordance with our contributed work on the project and our plan for the rest of the tasks until the FINAL. The specific dates and the number of days spent on each task including the names of the tasks can be found below.



4.2. Continuous Integration, Delivery and Deployment Plan

To support Continuous Integration automated testing is done. The most basic one is the unit test, which tests directly the functions and methods every time the project is built. In Java this is done with JUnit and in NodeJS we are using mocha. As an additional test specially for micro services we have integration tests, for that purpose we have a own testing server which tests the endpoints of our http (REST) Application. Every change will be committed to our Git Repository, so at every test, our code base is up to date. There the team has to trust in each member to commit only code which passes the tests and dont work on a side branch.



Continuous deployment/delivery is done by the containerisation with docker. Every new build can be deployed to production. In continuous deployment the docker is automatically created without user input.

Gitlab provides a full automatic CI/CD pipeline to finally deploy to production as a docker container. CI/CD concepts | GitLab

This can be an alternative to work with, but more research is needed here.

4.3. Distribution of Work and Efforts

Contribution of Member Kevin Miller:
Report:
- Chapter 1 (aside from bounded context diagrams)
- Logical View (every team member modeled their own services in detail)

Development:
- Initial design of system architecture
- Tutorial for initialization of new ms projects
- Reference implementation for microservices
- Reference implementation for Gateway RestController-classes
- Publish-Subscribe pattern & registration
- TransactionHistory Service
- Shipment Service

SR’s to be realized:
- Transaction History (estimated time remaining: 6h)
- Shipment (estimated time remaining: 8h)
- Gateway refinement (estimated time remaining: 8h)

Contribution of Member David Cömert:
Report:
- Chapter 4.2
- Chapter 5
- Physical Layer

Development:
- Cart Service
- Checkout Service
- Frontend (Mockup)
- (Manual) Integration testing and API development (End to End Tests)
- Prototype implementation microservices and connection with the Gateway

SR’s to be realized:
- Cart Service (estimated time remaining: 4h)
- Checkout Service (estimated time remaining: 10h)
- Gateway updates (estimated time remaining: 3h)

Further work:
- Create a integration test suite or server (estimated time remaining: 3h)
- Create ende to end integration tests for the services (estimated time remaining: 8h)

Contribution of Member Dorina Karameti:
Report:
- Chapter 2
- Chapter 4.1
- Scenario View (Use Case Descriptions) + Logical view: Notification Service, Rating Service

Development:
- Notification Service
- Rating Service

SR's to be realized:
- Notification Service (estimated time remaining: 8h)
- Rating Service (estimated time remaining: 6h)

Contribution of Member Tobias Fühles:
Report:
- Chapter 2
- Development View + Logical view: Account Service, Inventory Service
- Bounded Contexts

Development:
- MS Account Service
- MS Inventory Service
- Publish-Subscribe pattern & registration
- Gateway integration of services

SR's to be realized:
- Account Service (estimated time remaining: 4h)
- Inventory Service (estimated time remaining: 9h)
- Gateway Changes (estimated time remaining: 3h)

Contribution of Member Lukas Sorer:
Report:
- Chapter 1: Language context, Design Decision 8
- Chapter 3: Use-case diagram, Logical view: Price Crawler Service, Marked Product Service, Process Views

Development:
- Marked product Service
- Price adjustment Service

SR's to be realized:
- Marked product (estimated time remaining: 4h)
- Price crawler (estimated time remaining: 8h)

Task	Kevin Miller	Lukas Sorer	David Cömert	Tobias Fühles	Dorina Karameti
Discuss Design/Technology Stack	2	2	2	2	2
Weekly meetings	13	13	13	13	13
Concretization of System Architecture	4		2		
Create system requirements	1	1	1	1	1
Initial Draft Diagram Development View				3	
Refinement Diagram Development View				3	
Initial Diagram Logical View	6				
Refinement Diagram Logical View	6		1		
Initial Diagram Process View		2			
Refinement Diagram Process View		5			
Initial Diagram Scenario View					3
Refinement Diagram Scenario View					4
Initial Diagram Physical View			1		
Refinement Diagram Physical View			1		
Architectural overview class diagrams	4	1	1	1	1
Individual class diagrams	1	1	1	1	1
Document Bounded/Language Contexts	2	5	1	5	1
Use-case diagram		2			

Task	Kevin Miller	Lukas Sorer	David Cömert	Tobias Fühles	Dorina Karameti
Set up github	0.5				
Account Service				4	
Inventory Service				5	
Cart Service			2		
Checkout Service			4		
Frontend and Mockup	1	1	8	1	1
TransactionHistory Service	6				
Shipment Service	4				
Notification Service					
Price Crawler Service		1			
Rating Service					
Marked Product Service		2			
Gateway Service	12	2	5	2	2
Publish/Subscribe Pattern	5			4	
End to End Integration and Testing	1	1	12	1	1

5. How-to / mock-up documentation

Each service e.g. each bounded context has the defined http endpoints where the functions can be called. On startup they connect to the gateway service and subscribe to their endpoints. These endpoints can be called from a http client. In our case we have a front end based on NodeJs and HTML.

This user interface supports in the first step all main use cases. In addition the functional requirements from 2.1 are created..
Some services are further developed than others, which results in a half working ui. Some services need some investment to truly work with the ui, these ones are mocked and the ui is filled with simulated data, the others work as desired.
All services communicate event driven over the gateway, by subscribing the paths on the gateway. So they get an event if one of their paths are called.

5.1. Building guide

The Java Services are built with Maven as a Spring project. You may have to add “spring-boot:run” to your build options.
Each folder in /implementation is a service. At first it's necessary to start the gateway service. After that the order doesn't matter. They connect to each other on localhost with different ports.
The NodeJs environment is already dockerized, the building command is:

```
../cart $ docker-compose build
../cart $ docker-compose run -e GATEWAYIP=172.20.112.1 --service-ports app
```

where the *GATEWAYIP* is the currently ip where the gateway runs (not localhost)

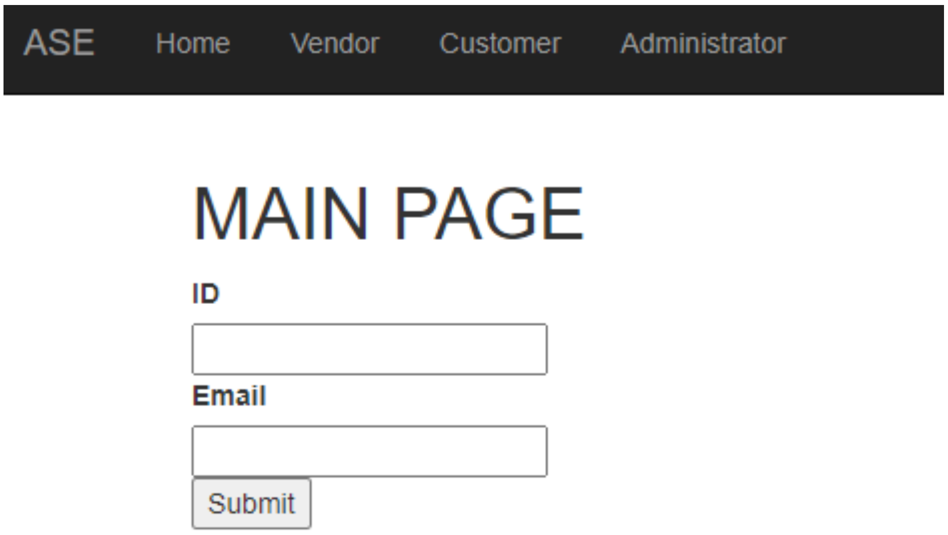
The same commands are necessary to start the frontend (mockup)

```
../frontend $ docker-compose build
../frontend $ docker-compose run -e GATEWAYIP=172.20.112.1 --service-ports app
```

5.1. Use case mockup

To connect with the frontend you need a modern browser and go to http://localhost:3003.

You will see the following:



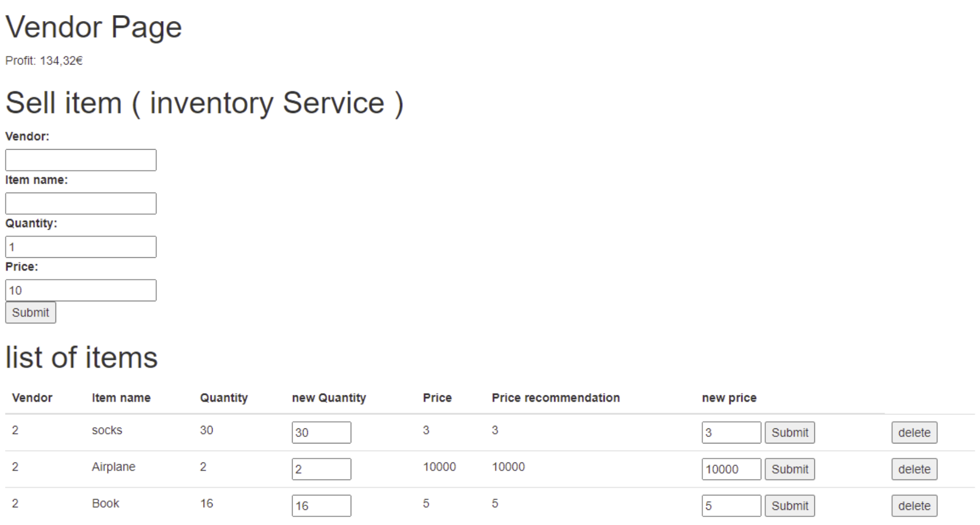
MAIN PAGE

ID

Email

Submit

With Id=1 you get logged in as a customer, with id=2 as a vendor and with id=3 as an administrator.
The page links on the top are only for development and testing purposes.
This will result in a redirect to the specific overview.
As a vendor you will be redirected to the vendor page.



Vendor Page

Profit: 134,32€

Sell item (inventory Service)

Vendor:

Item name:

Quantity: 1

Price: 10

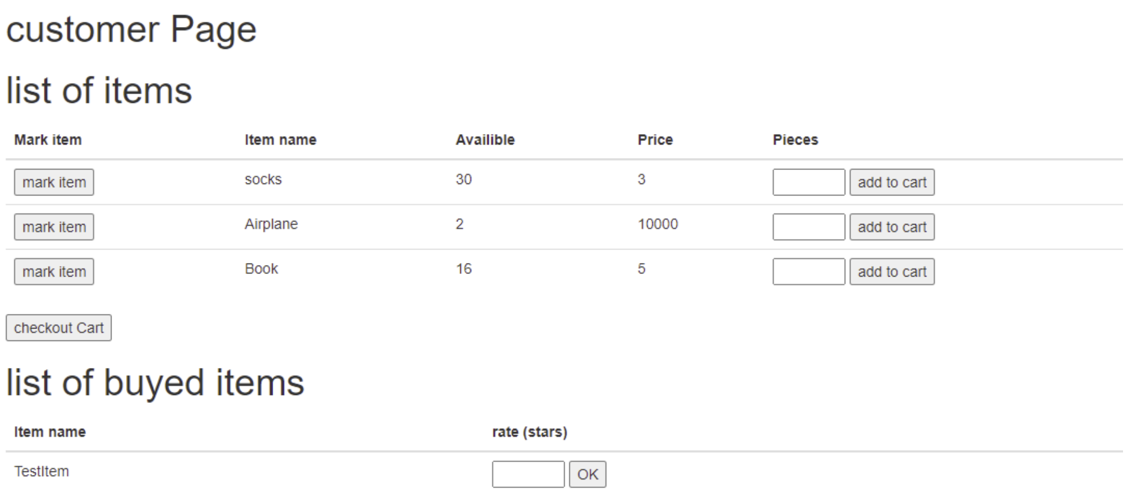
Submit

list of items

Vendor	Item name	Quantity	new Quantity	Price	Price recommendation	new price		
2	socks	30	30	3	3	3	Submit	delete
2	Airplane	2	2	10000	10000	10000	Submit	delete
2	Book	16	16	5	5	5	Submit	delete

Here you can add items with their properties. On the second table you see all items in the inventory which the vendors offer. These items can be manipulated as in the use case described.

If logged in as a customer, the following page will appear.


customer Page

list of items

Mark item	Item name	Availible	Price	Pieces
mark item	socks	30	3	add to cart
mark item	Airplane	2	10000	add to cart
mark item	Book	16	5	add to cart

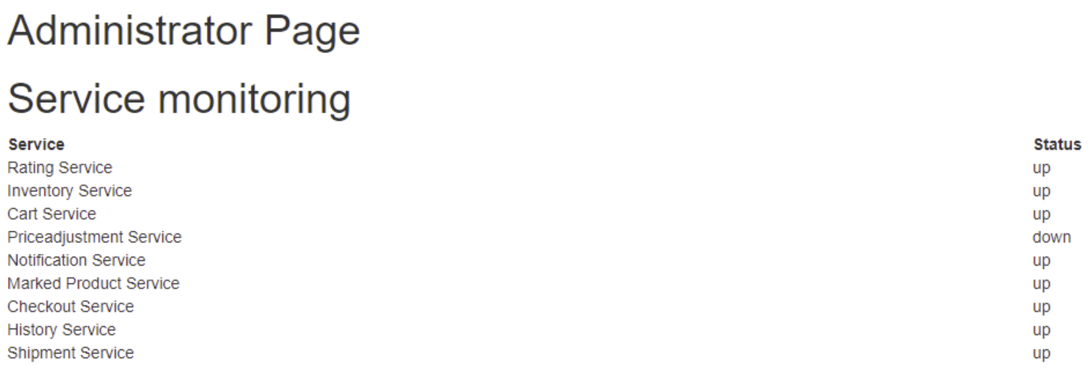
checkout Cart

list of buyed items

Item name	rate (stars)
TestItem	OK

Here all items from all vendors are displayed. The customer can now decide which item he/she wants to add to the cart. After this the checkout is possible.
The bought items can be rated.

At least the administrator view to monitor the services.


Administrator Page

Service monitoring

Service	Status
Rating Service	up
Inventory Service	up
Cart Service	up
Priceadjustment Service	down
Notification Service	up
Marked Product Service	up
Checkout Service	up
History Service	up
Shipment Service	up

In addition the administrator has access to the finished and latest builds of every service (e.g. from gitlab), so the administrator can start and deploy each service on his/her own.

Service & Endpoint Documentation

All projects (aside from the frontend) are Maven-based Spring projects written in Java and can be imported into any IDE and ran from there. No executables currently exist for DESIGN.

TransactionHistory

Endpoint	/generate
Method	GET
Input	/
Output	A list of transaction objects mapped to json. Returns a list of all transactions in the repository

Endpoint	/generate/{userid}
Method	GET
Input	userid of the customer/vendor that the history is requested for
Output	A list of transaction objects mapped to json. Returns a list of all transactions in the repository, that the user was involved in.

Endpoint	/generate/{type}/{userid}
Method	GET
Input	Type=role that the userid filled (“buyer” or “seller”); Userid=userid of the customer/vendor that the history is requested for
Output	A list of transaction objects mapped to json. Returns a list of all transactions in the repository, that the user was involved in in the respective role (type)

Endpoint	/add
Method	POST
Input	The transaction to be saved with fields: “customerid”, “vendorid”, “price”, “itemName”, “quantity”.
Output	The added transaction object

Shipment

Endpoint	/show
Method	GET
Input	/
Output	A list of all Shipment objects mapped json. Returns all shipments in the repository.

Endpoint	/show/{userid}
Method	GET
Input	/
Output	A list of all Shipments for the user with the respective userid.

Endpoint	/show/item/{itemid}
Method	GET
Input	/
Output	A list of transaction objects mapped to json. Returns a list of all transactions in the repository

Endpoint	/add
Method	POST
Input	A shipment object with fields: „customerId“, „vendorId“, „itemid“, „email“, „price”, “itemName”, “quantity”.
Output	The added Shipment object.

Endpoint	/start
Method	GET
Input	/
Output	/

	Starts the Shipment thread, which causes it to randomly dispatch shipments.
Endpoint	/stop
Method	GET
Input	/
Output	/
	Stops the Shipment thread.

Cart

Endpoint	**/getCart/{userid}**
Method	GET
Input	
Output	`{ "customerId": 1, "items": [ “itemId”: '2', “itemName”: 'Airplane', “quantity”: '1', “price”: '10000', “vendorId”: '2', “priceRecommendation”: '10000' ] }`
	Only if this item was added to the Cart before

Endpoint	**/deleteCart/{userid}**
Method	GET
Input	
Output	“OK”
Anmerkungen	

Endpoint	**/addItem/{userid}**
Method	POST

Input	“itemId”: '2', “itemName”: 'Airplane', “quantity”: '1', “price”: '10000', “vendorId”: '2', “priceRecommendation”: '10000'
Output	“OK”

Notification

Endpoint	/add
Method	POST
Input	{ "customerId" : "1", "itemId": "1", "price": "10", "itemName": "Item1", "shippingStatus": "delivered", "email": "john@email.com"}
Output	{ "customerId" : "1", "itemId": "1", "price": "10", "itemName": "Item1", "shippingStatus": "delivered", "email": "john@email.com"}
Anmerkungen	Shipping status can also be NULL depending on the type of notification

Endpoint	/shipping/{itemId}
Method	GET
Input	itemid
Output	Current shipping status of the item with the provided id
Anmerkungen	

Endpoint	/price/{itemId}/{newPrice}
Method	GET
Input	Itemid, newPrice
Output	The old price of the item with the given id and the new price
Anmerkungen	

Endpoint	/clearAll
Method	GET
Input	
Output	
Anmerkungen	Clears all items in the repository

Inventory Service

Endpoint	/
Method	POST
Input	[{ "price": 22.0, "vendorId": 30, "quantity": 2377, "itemName": "socks", "priceRecommendation":23.4 }]
Output	[{ "itemId": 70, "price": 22.0, "vendorId": 30, "quantity": 2377, "itemName": "socks", "priceRecommendation": 23.4 }]
Anmerkungen	itemId is auto generated with Spring annotation

Endpoint	/{id}
Method	GET
Params	id=70
Output	[{ "itemId": 70, "price": 22.0, "vendorId": 30, "quantity": 2377, "itemName": "socks",

	"priceRecommendation": 23.4 }]
Anmerkungen	Returned as JSON. Single item searched via Id.

Endpoint	/items/
Method	GET
Input	
Output	[{ "itemId" : 69, "price" : 2232.0, "vendorId" : 30, "quantity" : 2377, "itemName" : “test”, "priceRecommendation" : 70 }, { "itemId" : 70, "price" : 22.0, "vendorId" : 31, "quantity" : 2377, "itemName" : "socks", "priceRecommendation" : 23.4 }]
Anmerkungen	Returned as JSON. All items returned

Endpoint	/vendor/{id}
Method	GET
Params	E.g. id=60
Output	[{ "itemId" : 69, "price" : 2232.0, "vendorId" : 60, "quantity" : 2377, "itemName" : “test”, "priceRecommendation" : 70 }, { "itemId" : 70, "price" : 22.0, "vendorId" : 60, "quantity" : 2377, "itemName" : "socks", "priceRecommendation" : 23.4 }]
Anmerkungen	Returned as JSON. All items returned for specific vendorId.

Endpoint	/update/{id}
Method	GET
Params	E.g. id=69
Input	[{ "quantity" : 2377}]
Output	[{ "itemId" : 69, "price" : 2232.0, "vendorId" : 123, "quantity" : 2377, "itemName" : “test”, "priceRecommendation" : 70 }]
Anmerkungen	Updates a single Item by id

Endpoint	/delete/{id}
Method	GET
Params	E.g. id=69
Output	NULL or ‘Delete successfull’
Anmerkungen	Deletes a single item by id.

Marked Product

In order to test the applications properly I would recommend for the marked product service to first execute some /mark POST-requests as it is the function that writes to the repository and almost every other function depends on some data in the repository. So first execute 1-3 /mark requests and then test the other endpoints. Have fun!

Endpoint	**/show/{userid}**
Method	GET
Input	
Output	[{ "id": 1, "customerId": 111, "vendorId": 333, "price": 131.7, "email": "max.muller@gmail.com", "itemName": "Samsung A5" }

	]
Anmerkungen	userid: can be customerId or vendorId To receive the output above {userid} has to be either 111 or 333

Endpoint	**/showall**
Method	GET
Input	

Output	[
	  {
	    "id": 1,
	    "customerId": 111,
	    "vendorId": 333,
	    "price": 131.7,
	    "email": "max.muller@gmail.com",
	    "itemName": "Samsung A5"
	  },
	  {
	    "id": 2,
	    "customerId": 111,
	    "vendorId": 444,
	    "price": 831.7,
	    "email": "max.muller@gmail.com",
	    "itemName": "IPhone XS"
	  }
	]

Anmerkungen	shows all marked products and with the corresponding attributes as seen above

Endpoint	**/update**
Method	POST

Input	{
	  "customerId": "111",
	  "vendorId": "444",
	  "price": "841.7",
	  "itemName": "IPhone XS",
	  "email": "max.muller@gmail.com"
	}

Output	
Anmerkungen	Don’t know if necessary - product related updates should happen in the inventory service?

Endpoint	**/mark**
Method	POST
Input	{ "customerId": "222", "vendorId": "555", "price": "212.87", "itemName": "Huawei P20 lite", "email": "lisa.hofer@hotmail.com" }
Output	{ "id": 3, "customerId": 222, "vendorId": 555, "price": 212.87, "email": "lisa.hofer@hotmail.com", "itemName": "Huawei P20 lite" }
Anmerkungen	This is the actual marking of items, when executed it will be stored in the markedproduct repository

Endpoint	**/print**
Method	GET
Input	
Output	
Anmerkungen	Prints all marked products into the console, based upon the examples from above it would look like this: -- [itemid: 1; Buyer: 111; Seller: 333 |Samsung A5, price: 131.7€|] [itemid: 2; Buyer: 111; Seller: 444 |IPhone XS, price: 831.7€|] [itemid: 3; Buyer: 222; Seller: 555 |Huawei P20 lite, price: 212.87€|] --

Price Adjustment

Just send a random string in a JSON format to the /recommend endpoint as a POST request and receive a random double as result. Due to the fact that we should only provide some mock-up functionality for this DESIGN-submission the real logic of price crawling is not implemented yet.

Endpoint	/recommend
Method	POST

Input	{
	    “itemName”:”IPhone 10”
	}

Output	85.312312
Anmerkungen	Input can be any String

Rating

To test the rating service, start by adding a new item and rating in a JSON format as displayed in the “/add” endpoint. After this step you can locate the getRating endpoint to get the current rating of the added product and rating. The checkRatings endpoint returns all ratings currently in the repository.

Endpoint	/add
Method	POST

Input	{ "customerid": 1,
	     "itemId" : "1",
	      "itemName": "Item1",
	      "rating": "1"}

Output	{ "customerid": 1,
	     "itemId" : "1",
	      "itemName": "Item1",
	      "rating": "1"}

Anmerkungen	

Endpoint	/getRating/{itemId}
Method	GET
Input	itemid
Output	The current rating of the item
Anmerkungen	Rating is returned as a double

Endpoint	/checkRatings
Method	GET
Input	

Output	{ "customerid": 1,
	     "itemId" : "1",
	      "itemName": "Item1",
	      "rating": "1"}

Anmerkungen	

Checkout

Endpoint	**/checkout/{userid}**
Method	GET
Input	

Output	{
	 "command": “delete”,
	 }

Anmerkungen	

Account Service

Be aware that this service depends on the Transaction History.

Endpoint	/vendor/{id}
Method	GET
Params	E.g. id=60
Output	[{ 2345.6 }]
Anmerkungen	Returned as JSON. Single double value returned for revenue of specific vendor selected by vendorId.