DELFT UNIVERSITY OF TECHNOLOGY
GROUP 51

Assignment 1

08.01.2021

Task 1 - Software Architecture

1. Architecture & Interaction

Our system relies on the microservice architecture, which greatly helps in both scaling and maintaining our application; also the low-coupled components of the microservices make the application more adaptable for changes over time.

We have decided to split our application into three main microservices: authentication, requests and transactions.

How did we decide on these three specifically?

We tried to separate the functionalities as much as possible, and make the requirements as concise as possible, in order to have a better understanding on how we should structure our project. The authentication microservice was quite self explanatory, and the quickest one to figure out - we needed a way to register, authenticate our users and grant them access to our application.

The next step was grouping the requirements that have something in common, observing the results, and thinking on how exactly we could separate them into multiple microservices, if they were needed. We had a good overview of what our application should do: users are part of a household, they have a common fridge, where the products are shared between the members, one user per household can go and buy groceries at a time, and credits are subtracted or added based on the products being bought or used.

From this point onward, we understood we need a way to store multiple houses which users can join, and a way to keep track of the products in the fridge, and the transactions of each user, when it is their turn to go grocery shopping. This is how we divided these requirement for two more microservices: requests (user joining/leaving house, sending requests to enter a house, members granting approvals to enter) and transactions (buying/consuming food from the fridge, keeping track on which products were used by which members, updating/splitting credits of the users as they take products from the common fridge).

These three microservices interact with each other through Eureka, whose purpose is to establish the communication between microservices, and the Gateway, which checks a given JWT token, where requests will pass on to other microservices if the token is a valid and signed one.

We have chosen to send the token with each request that is being sent, as it is a secure way to ensure our system will not be easily broken into, and malicious users will have a hard time in impersonating others users, in the case that they try to send requests with different usernames they may have found.

In order to implement these microservices we have made use of several technologies, which combined lead us to the desired result. Each microservice makes use of the properties of Spring in order to make possible handling requests and sending responses. Each relevant entity, which represents a database table, has a corresponding Controller class that listens to the requests generated by users and uses JPA repositories to use queries written for that specific entity. These queries are executed externally on the database and sends back tuples as responses which are than handled by Spring. This process can be observed in Figure 1. We have made use of the features offered by Gradle in order to build the project and handle dependencies. Although we have had a file containing general dependencies needed in all of the microservices, in order to reduce code duplication, we also have gradle files implemented in each microservice with specific dependencies needed in creating the microservice. Furthermore, our project makes use of three MySQL databases, one for each microservice.

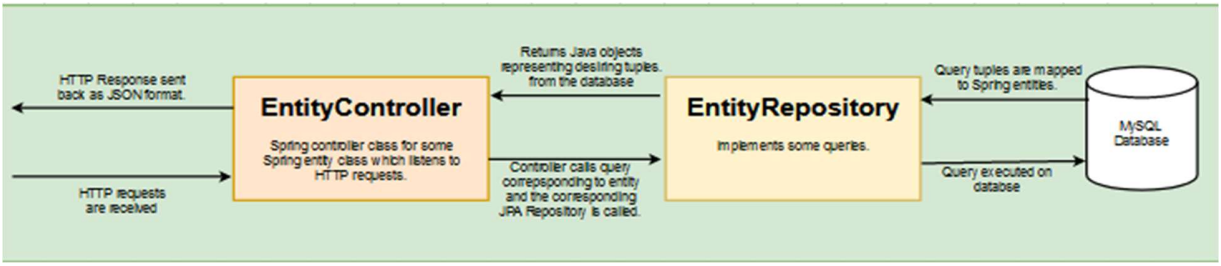


Figure 1: Inner working of Spring controllers and repositories in handling HTTP requests

2. Microservices

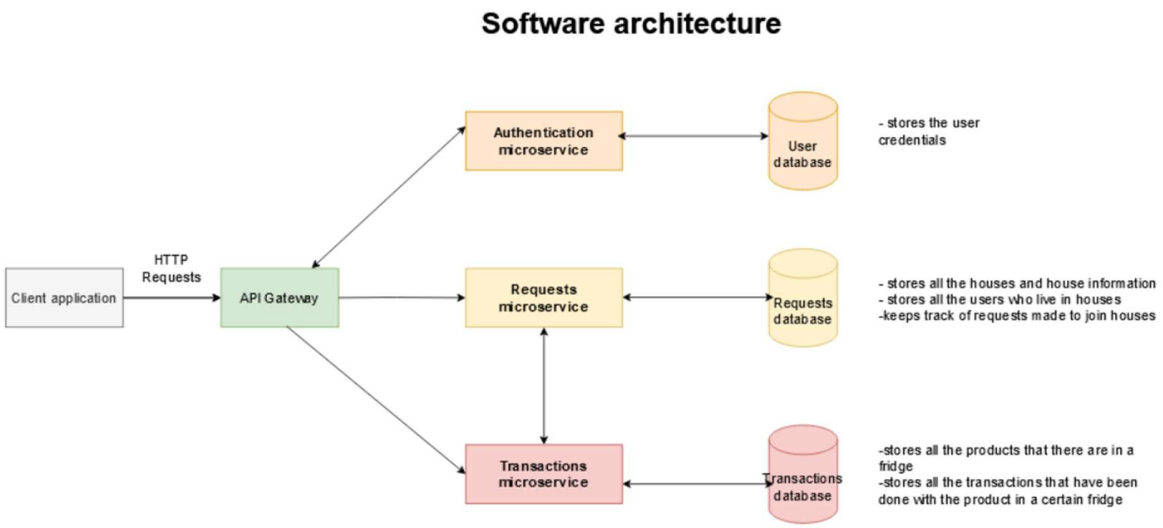


Figure 2: Component diagram of our system

2.1. Gateway and Eureka

Our system makes use of two separate microservices in order to establish the communication between the microservices in an efficient way. Eureka is usually considered to be a load balance, but we have made use of it in order to register each microservice, as it allows microservices to find, and therefore, communicate, with each other. In order to create these two microservices we have made use of Netflix's open source library Zuul and Eureka. The Gateway represents the only microservice that users will make requests to. Furthermore, in order to pass requests to the other microservices, it requires a JWT token to be sent through with every request in order to verify the identity of a client. In case the user sends a valid and signed token, the gateway will redirect the client to the desired microservice. Figure 3 depicts the process that the Gateway goes through when a request is sent.

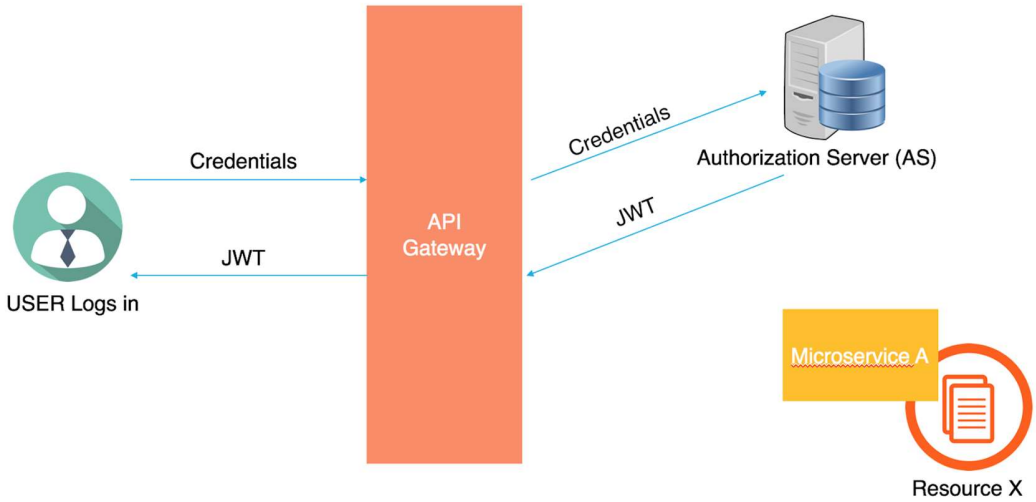


Figure 3: The interaction between the Gateway and the user.*

2.2. Authentication

The authentication microservice is the one to be accessed by the Gateway in case an invalid token is sent through and the user is not logged in or registered, as it remains the only microservice to which the user can send POST requests in order to request registration or authentication. After the Gateway receives the invalid token and an authentication or registration request, it passes it on to Authentication which checks user credentials and adds them to the database in case of a registration. If the credentials are valid, it signs a JWT token and sends it back to the Gateway for the further use of the other microservices.

2.3. Requests

The requests microservice is the one which mainly handles requests of users joining a house and members receiving and approving the requests directed to their own household. Since we had to keep track of the users, the houses available and the requests sent by the users, this microservice contains 3 entities: User, House and Request; each with their own important attributes. In the current microservice there are of course controllers for our entities, more specifically the User, House and Requests Controllers. There exists several other functionalities available, besides the ones stated above, and they mainly revolve around the credit status of the users, splitting the credits of users, updating them and so on. These functionalities are crucial for the Transactions Microservice, when users use certain products and the credits will need to be updated. Since the Requests microservice has access to the user table in the database, the credits as one of the attributes, we have created a communication between the Requests and Transactions Microservices, thus the credits of the users could then be updated successfully.

This microservice is dependent on the authentication microservice which signs a JWT token when a user successfully enters the system. Furthermore, it is also dependent on Eureka, where the communication is established between the microservices, and on the Gateway, where a user is verified in the system, through the given JWT token, which will then be passed to the Requests microservice, if they have been granted access.

2.4. Transactions

Lastly, the Transactions microservice is the one which mostly handles all the tasks related to buying and consuming food. As a first step, a user is able to add products to the common fridge as well as delete or edit his products in case a mistake has been made. Furthermore, each user is able to add transactions, which means each user is able to indicate every time he takes a certain portion of food from the fridge. All these features also entail some changes in the credits that a user might have. Whenever a user adds a product, credits equal to the price of the product are added to his account and everytime he decides to consume something, a certain amount of credits, equal to the price per portion, is subtracted from his total number of credits. We have implemented these functionalities of juggling with credits by adding a Microservice communicator class to our Transaction microservice which implements a couple of functions especially designed to send requests to the Requests microservice.

On the other hand, the Transactions microservice communicates with Eureka and Gateway. As already mentioned, Eureka establishes the proper communication between different microservices while the Gateway checks whether a user has a proper JWT token. In this way, the latter makes sure the system is safe to use and no unidentified user can enter the system.

3. Conclusions

After seeing the functionalities and the structure of our system, we can conclude that we have implemented a system based on the microservice architecture pattern. The decision of splitting our work into three main microservices was taken based on the prioritization of requirements and their aim. Therefore, our first microservice, the Authentication microservice, is the only one which deals with user registration and authentication. We have opted to implement a separate Requests microservice, which only deals with problems regarding users and their appartenance to a house. The Transactions microservice communicates with the previously mentioned microservice, but is totally independent and is the only one which deals with everything that concerns products. These three microservices are able to communicate and harmoniously work together thanks to the Gateway service, which is the one handling all requests and verifying the identity of users, and Eureka service, which makes sure microservices are able to find each other.

To sum up, our system is based on the microservice architecture which enabled us to work more efficiently, especially with issues that might have risen, as well as, makes it easier to continuously update and maintain such a system.

List of references

(1) https://github.com/Netflix/zuul
(2) https://github.com/Netflix/eureka
(3) * https://sdtimes.com/wp-content/uploads/2017/09/image2.png
(4) Microservice tutorial: Architecture and example, https://www.guru99.com/microservices-tutorial.html#4
(5) Microservices with Spring, https://spring.io/blog/2015/07/14/microservices-with-spring

Task 2 - Design patterns

I. “Strategy” design pattern

1) Description of why and how the pattern is implemented:

We have decided to implement the Strategy design pattern. We want to get a list of all products from the database. There are many possible orders of these products so we want to let the client decide whether they want to get the cheapest products or the ones that have the most portions first.

To solve this we could put each possible sorting method in different functions, unfortunately that would create a lot of code duplication. On the other hand if we wanted to put all of the options in a single method to avoid duplication, it would end up neither maintainable nor readable.

Having a conditional with many such functionalities would be almost impossible to change in parallel with other developers and would create many merge conflicts.

Strategy design pattern solves this by separating the context (code where sorting is used) from the behaviour (actual sorting strategy). We create an interface class *SortProductsStrategy*, which is implemented by the following classes: *Price Strategy*, *NameStrategy*, *AmountStrategy*, *PriceThenAmountThenNameStrategy* and *RandomStrategy*. This interface acts as an API between the *ProductController* and concrete sorting strategies. By dependency injection, the product controller receives an object which has the overridden method sortProducts() which sorts the products, based on the strategy the user has deemed appropriate.

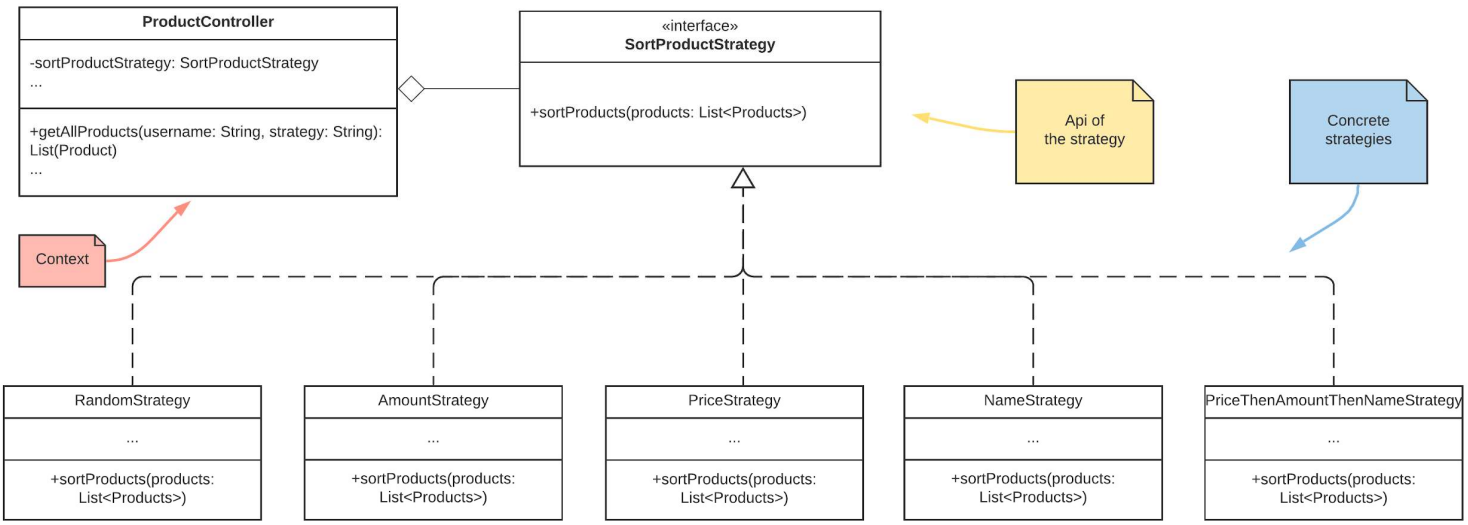
Dependency injection additionally makes unit testing more appropriate because we can mock the strategy and test only the core functionality of the product controller. The sorting strategies can be tested separately from the controller, nicely splitting the tests into two separate independent units, which makes spotting bugs much easier.

By splitting behaviour (concrete sorting strategies) from context (getAllProducts() method in ProductController) we are distributing the complexity among many classes. Without this distribution, the method in the ProductController would be very complex (cyclomatic complexity would reach 9). This separation also makes it possible to change the behaviours of the strategy without knowing (or even looking at) the ProductController.

This makes the following situation possible: Senior developer who is responsible for maintaining ProductController asks an intern to change the behaviour of one of the strategies. The intern can do it without knowing how the product controller works. Which

makes the senior developer not wasting the time on explanations and interns not overwhelmed by the amount of code to understand at once.

2) Class diagram of how the pattern is structured in our code:



3) Implementation of our design pattern:

"Context" - getAllProducts() in ProductContoller:

```
@GetMapping("/allProducts")
public @ResponseBody
List<Product> getAllProducts(@Username String username,
                             @RequestParam String strategy) {

    System.out.println(username);
    List<Product> products = productRepository.findAll();

    if (strategy == null) {
        sortProductsStrategy = new RandomStrategy();
    } else {
        switch (strategy) {
            case "amount":
                sortProductsStrategy = new AmountStrategy();
                break;
            case "name":
                sortProductsStrategy = new NameStrategy();
                break;
            case "price":
                sortProductsStrategy = new PriceStrategy();
                break;
            case "priceThenAmountThenName":
                sortProductsStrategy = new PriceThenAmountThenNameStrategy();
                break;
            default:
                sortProductsStrategy = new RandomStrategy();
        }
    }

    sortProductsStrategy.sortProducts(products);
    return products;
}
```

“API” communication between controller and concrete strategies - SortProductStrategy:

```java
/**
 * Interface that provides a blueprint for different sorting strategies of products.
 * Introduced to adhere to "strategy" design pattern.
 */
public interface SortProductsStrategy {
    /**
     * Sorts products.
     *
     * @param products products to be sorted
     */
    void sortProducts(List<Product> products);
}
```

“Behaviour” (strategy for sorting) - on the example of PriceThenAmountThenNameStrategy:

```java
package nl.tudelft.sem.transactions.strategy;

import java.util.List;
import nl.tudelft.sem.transactions.entities.Product;

/**
 * Provides functionality for sorting products primarily on their price,
 * secondarily on number of portions left, ternary on their name.
 * Implements SortProductsStrategy to adhere to "strategy" design pattern.
 */
public class PriceThenAmountThenNameStrategy implements SortProductsStrategy {
    /**
     * Sorts products primarily on their price,
     * secondarily on number of portions left, ternary on their name,
     * all in ascending order.
     *
     * @param products products to be sorted
     */
    @Override
    public void sortProducts(List<Product> products) {
        products.sort((o1, o2) -> {
            if (o1.getPrice() == o2.getPrice()) {
                if (o1.getPortionsLeft() == o2.getPortionsLeft()) {
                    return o1.getProductName().compareTo(o2.getProductName());
                }
                return Integer.compare(o1.getPortionsLeft(), o2.getPortionsLeft());
            }
            return Double.compare(o1.getPrice(), o2.getPrice());
        });
    }
}
```

We also wrote new tests for newly implemented methods.

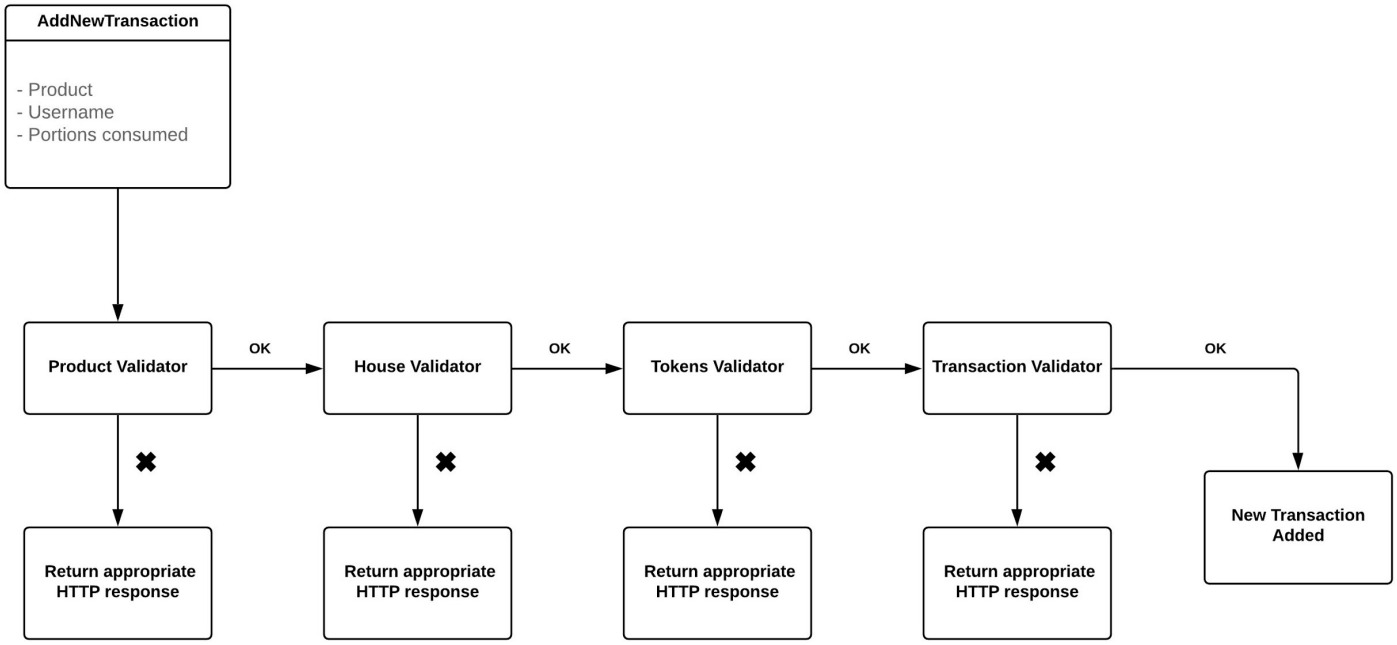
II. Chain of responsibility design pattern

We discussed the chain of responsibility design pattern as a group and figured that we use something like that with our authentication microservice. We have implemented a gateway that automatically checks if a user is authenticated based on a JWT token sent with every client request. We decided that this is not enough, because we did not have enough code to show, as most of this functionality was implemented by the gateway library we used. Therefore, we extracted most of the transaction logic in our application in several validators. This helped us with some of the code duplication in the transactions microservice, because we used the validators in two methods that did almost the same thing.

1) Description of why and how the pattern is implemented:

The transaction operation is one of the main operations in our application. It enables a user to get food from the fridge of his house. One of the main disadvantages of our implementation was that we assumed that the input provided by a user would always be correct. This approach however, is very error prone and this inspired us to add new checks for the information provided by the user, when they want to make a new transaction. In order to implement a verification of the user input we decided to use the Chain of Responsibility design pattern. It ensures that we will not create overcomplicated methods with high cyclomatic complexity and that it will be a lot easier to add more checks by simply creating new validators.

Figure 1:



On Figure 1 you can see a diagram that simply represents the architecture of our Chain of Responsibility design pattern.

When a user makes a request to add a new transaction (meaning they decide to take some food from the fridge, or share food with a roommate), first the product validator is called. It ensures that a product with the given ID exists in the database, there are enough portions, and that the product is not expired. If some of these checks fail, the product validator returns the appropriate HTTP Response entity. Otherwise, the product validator calls the next validator in the chain.

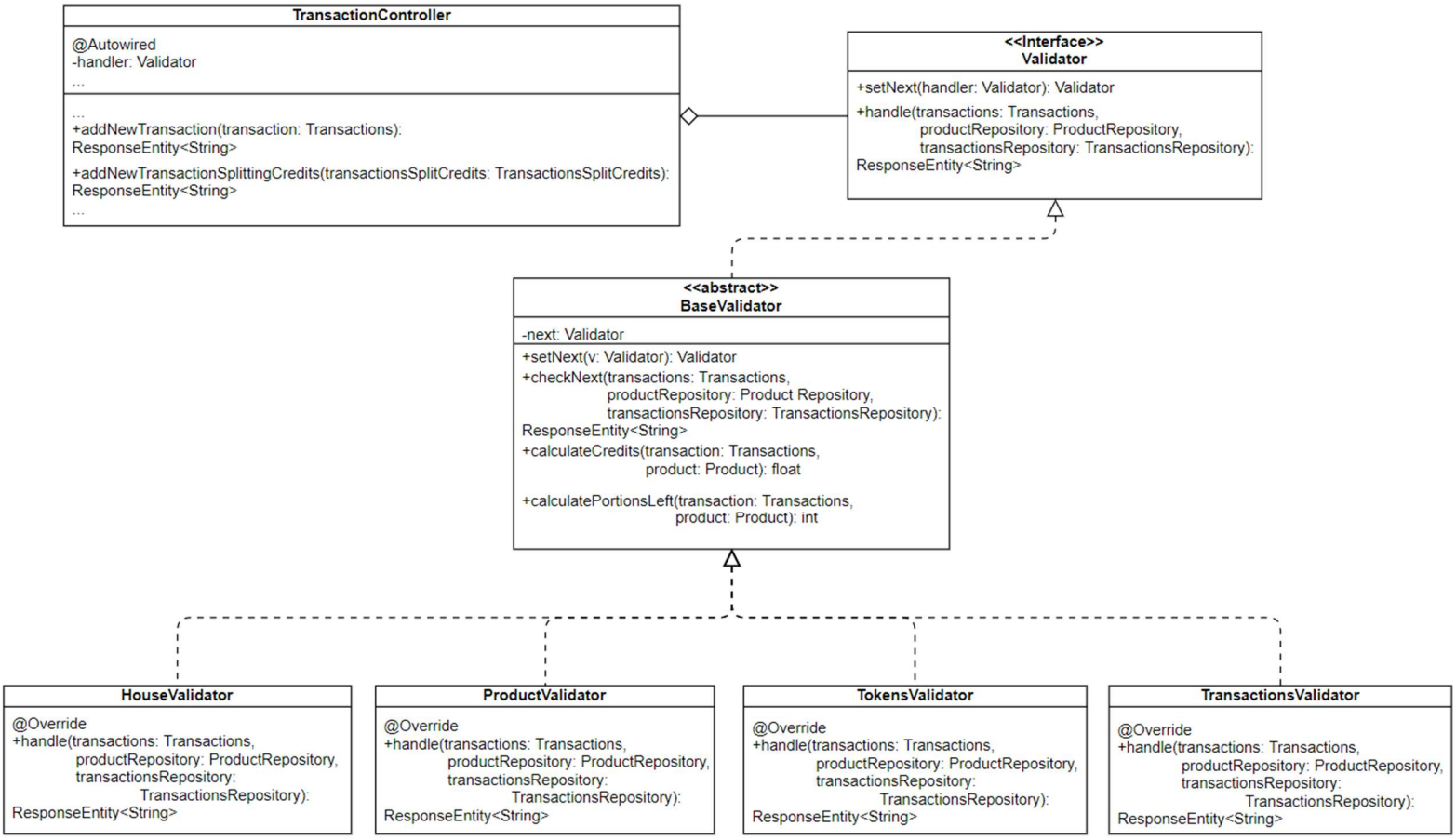
The next validator in the chain is the house validator. It checks if the user, making the request and the product belong to the same house. Otherwise, users could eat products from other houses, which we decided in the beginning is not a desired functionality. Again, if some of the checks fail, the validator returns.

However, if everything is in order, the next validator in the chain is the tokens validator. It makes sure that there is a change in the credits. Otherwise, there is no need to continue, because there must have been some mistake in the request, so the validator returns.

If there has been no such error in the request, the last validator we have implemented is the transactions validator. It is probably the most important validator, because there we execute the change of credits of a user or group of users. Therefore, we needed it to be last, because it changes the state of the credits in the requests microservice's database, so we first had to ensure that the change is indeed valid. We save the transaction, update the product and user information and return the appropriate HTTP Response status and message.

The chain of responsibility design pattern simplifies the way to expand the functionality of the transactions logic. For example, we did not have the check if the user is in the same house as the product before, but it was really easy to add it, after we refactored the code. Since the validators are logically separate, it was enough to add a new validator class that implements the additional check and add this validator to the chain. We did not need to add anything to the existing validators.

2) Class diagram of how the pattern is structured in our code:



3) Implementation of our design pattern:

The design pattern is implemented in the transactions microservice. We added a new folder “handlers/” that contains all the validators including an interface and an abstract class for a base validator.

Since we use the validator chain twice, we decided to make it a Bean, and we implemented it in the entry point of the microservice - its Application.java file. Then, we autowire the validator in the Transaction controller, and use it in the *addNewTransaction* and *addNewTransactionSplittingCredits* methods.