SEM Group 50

Assignment 1

Task 1

Currently, we have two microservices - an Authentication microservice and a Food Management microservice. Both the Authentication and Food Management microservices operate on a separate server with a separate database. Our application also uses an API gateway.

The API Gateway is solely used as a proxy server that authenticates requests and reroutes them. In the source code of the project, we have created two classes - “AuthenticationCommunication” and “FoodCommunication” - they are the API that would be used by the front end of the application. if there was one. The idea is that the communication classes send HTTP requests to the API Gateway. Those requests first get authenticated and then get rerouted to the correct microservice, which in return sends back a response. Based on said response the Gateway microservice sends a response back to the aforementioned communication classes. The idea behind the API Gateway is that we needed to create something that would facilitate the communication between the user application and the other microservices, whilst providing authentication as well, instead of having the whole authentication in the Authentication microservice, as it would become too clustered, and authenticating users would become too troublesome (a request sent from an unauthorized user would have to go through the gateway, then to the authentication microservice where it would be authenticated, and then back to the gateway in order to be rerouted to the correct destination microservice). In order to further secure the application, we implemented our endpoints in such a way that a request cannot be sent directly to a microservice and conduct an operation, without first going through the gateway. The idea is that when an endpoint receives a HTTP request, it checks whether that request contains a JWT in the header. If that is the case, then a HTTP request is sent to the gateway with said token in order for it to be validated. If the token is successfully validated, in other words if it is a valid token, then the user is granted access to the desired functionality. If the token happens to be missing from the request header or is invalid, the user is not granted access to the desired functionality. Below is a description of how the JWT works.

The Gateway microservice uses JSON Web Tokens to authorize the users to the secured endpoints. First, the user has to login with his/her credentials and they are compared to those stored in the database. If they are correct the application generates a JWT for the user. The token consists of three parts separated by dots. The first one is a header that stores information about the encryption algorithm, the second part is the payload. It contains information about the user such as the email or the token’s

expiration date. The first two parts are encoded using the Base64 algorithm this means that this data is decodable by the user. The last part, however, contains a signature encrypted using the SHA-512 algorithm using a prespecified secret string. A user can then identify him/herself by sending an additional HTTP header containing this token. There is no need for the user to end the sessions. Tokens by themselves are valid for 10 days, but If the user wants to make it unusable earlier, he/she needs to just remove it since tokens are only stored on the client side.

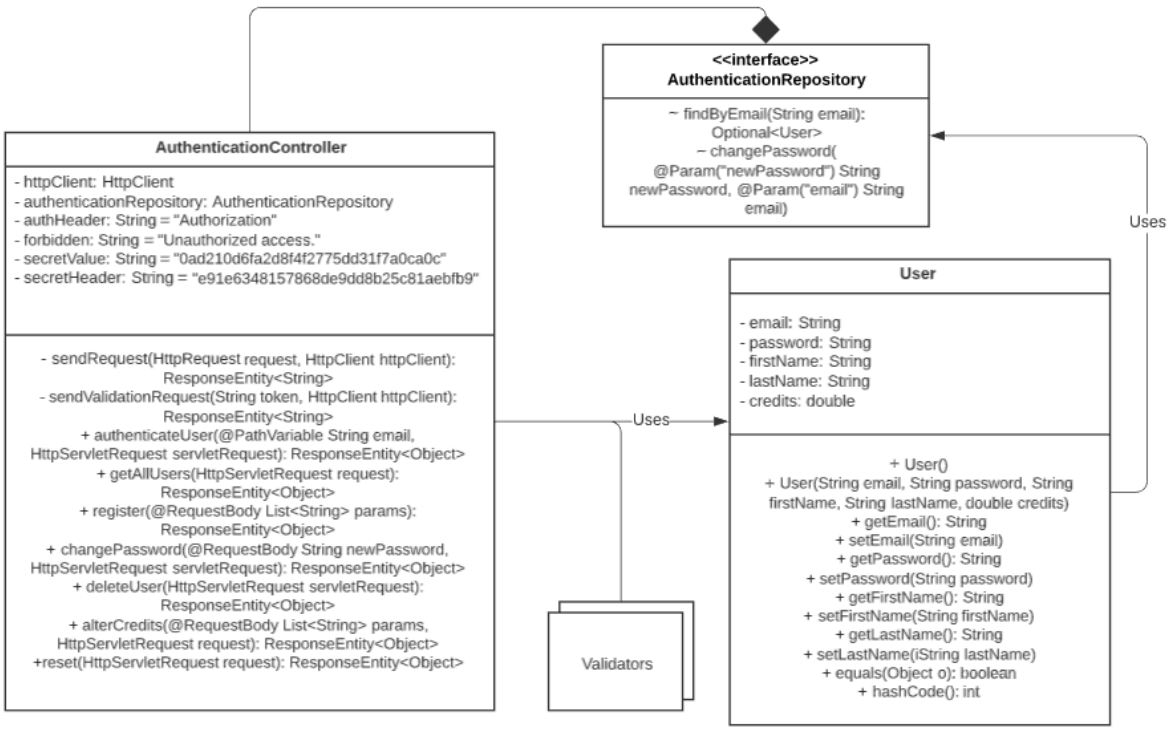
The Food Management microservice is responsible for storing information about food and handling food sharing. Users can post their food listings containing the number of portions they are offering to share and their respective costs. Then users in a group or by themselves can get products for credits they previously obtained. The Food Management and Authentication microservices exchange requests in order to check if a user has enough credits on his/her account and if so then transfer them to the food buyer account. The system also has special behaviour, such that if a food becomes spoiled its cost is split among all users because preventing food waste is a team effort.

The Authentication microservice is completely separate and is solely used for handling user data. Its responsibility is to handle and alter user data. It provides the following functionalities: creating a user account, a HTTP request with the user credentials - email, password, first name, and last name - is received. First, the validity of all said values is checked. If all values are valid, and an account with the given email doesn't exist, then the account is saved into the database, otherwise, a response with a bad request status is sent back to the gateway. The microservice also provides the ability for a user to change his/her password. A HTTP request with the new password is sent to the endpoint. The validity of the potential new password is checked, and then the password of the user is updated in the database. Otherwise, a response with a bad request status is returned. Moreover, a user can delete his/her account. Said endpoint simply deletes the account. A functionality that alters the credits of a user is also provided. It receives the number of credits that should be added to the balance of said user via a HTTP request and alters the value in the database. Last, but not least a reset functionality is provided. It resets the credits of all users in the database back to 0. As explained earlier all functionalities except the registering and logging in check whether there is a token in the header of the request and if it is such it is validated. If a token is not provided or an invalid token is provided, then a response containing a 403 status is returned, meaning that the user is not granted access to the following functionality.

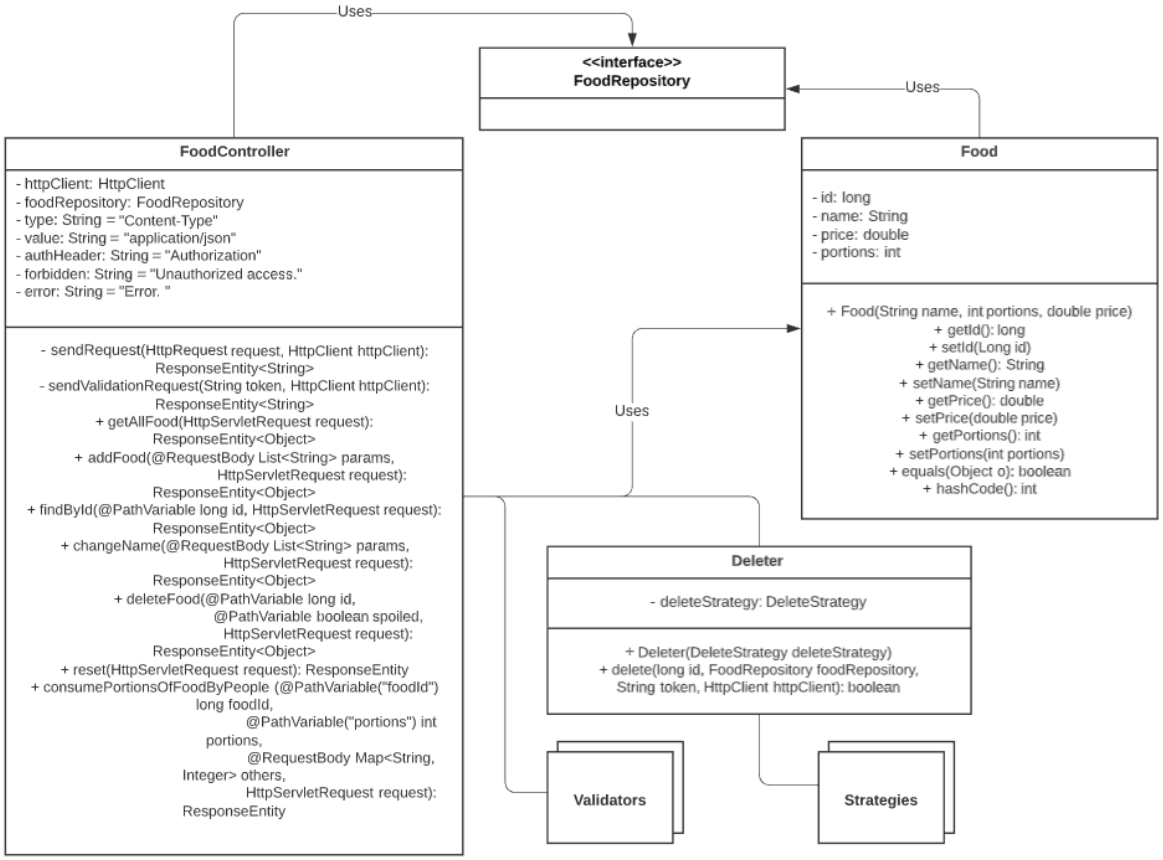
Our idea behind the two microservices was that each one of them would provide a different set of functionalities (related only to itself). The result of that would be better functionality isolation. Whenever a microservice needs functionality that is provided by another microservice, it sends a HTTP request to the gateway, and the gateway reroutes the request to the correct endpoint in the correct microservice. Hence, the

gateway is not only the single point of entry into the application but also the single point of communication within the backend. This provided us with scalability and abstracted communication, as there is no direct communication between the microservices. The idea behind this is that if a third microservice is added to the application, we wouldn't have to write additional methods in each microservice that handle the communication with the new microservice, but rather merely send a request to the gateway, which would reroute it to the new microservice. This makes the gateway a centralized point of communication, hence making the communication between microservices much easier to scale and alter later in development.

Authentication microservice



Food Management microservice



Task 2

Chain-of-responsibility design pattern

Why we chose this design pattern?

The chain-of-responsibility design pattern is known for its easy scalability and for reducing coupling. This means that it will be much easier to add extra validators if there is ever such a need. Also, this pattern delegates the verification functionality to other classes, which makes the main method and the respective class look cleaner. This leads to a better visualization of what is happening within the program without seeing the details of it in one method. Moreover, only one class is in charge of handling the chain, which makes it easier to use and alter if necessary.

The subclasses of the main handler offer the app more flexibility and make it much easier to test. Because the code is now divided over multiple classes, we can test these classes separately. Then when we want to test the main method, we only need to test the main functionality of this method, and not all the validation it does.
Multiple handlers are linked one after the other forming a chain, and the pattern allows them to process a request part by part. This way the request will be processed by all handlers. The loose coupling approach refers to the simplicity of the flow. The classes which process the requests only need to know that those requests are being handled “accordingly”. The easy reusability of the handlers for various methods helps with the reduction of code duplication. In the next paragraph we will be talking about the food management microservice, but we have also implemented this pattern for the authentication microservice in an analogous way.

The methods we applied this design pattern to, were very long. They also had many side effects, which all referred to the validation. This made these methods very difficult not only to test but also to understand. Moreover, int different scenarios different validation may be required - for example, if you only want to change the user’s password, you have to go through the entire validation chain. Therefore, we have decided to apply this design pattern, to make the code better structured, easier to test and easier to understand.

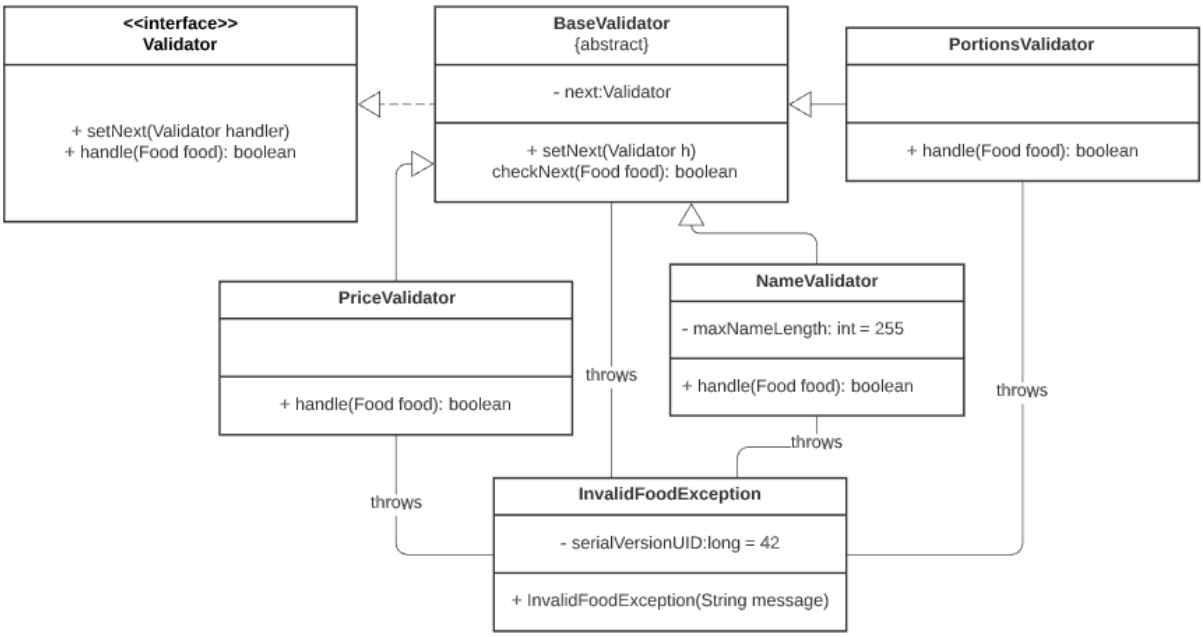
How we applied this design pattern?

The BaseValidator is the abstract class which implements the methods required by the Validator interface (setNext - defines the chain of handlers, and handle - processes a request). The children of the BaseValidator class are called in the FoodController or the AuthenticationController class and their function is to validate requests. In the case of the AuthenticationController these classes are: NameValidator - checks, if the food’s name is at most 255 characters long and whether it contains only the appropriate characters; PortionsValidator - checks, if the given number of portions is a positive integer; PriceValidator - checks, whether the given price is a nonnegative integer.

Whenever some number of portions of a food is added to the database, the required validators are initialized. We use the setNext method of all the validators in order to

chain then together in the appropriate way and order. Then the food object is parsed through all these validators. In case that any one of them raises an exception, the original method will also raise an exception and, therefore, the food will not be added to the database. If and only if all the validators pass and no exceptions are raised, the food will be successfully added to the database. Below is a diagram of the chain of responsibility design pattern.

Class diagram of the design pattern



Strategy design pattern

Why we chose this design pattern?

Our system uses the microservice architecture and is already complex enough. By using the Strategy design pattern, we aim at improving the way we structure our code and making it more comprehensible and flexible. To achieve this flexibility, we created the *Deleter* class. In the next paragraphs, we will focus on the strategies implemented for deleting food from the database. This design pattern is dedicated to eliminating the need for conditional statements by providing the ability to change the behaviour in run-time within a family of algorithms. Moreover, this design pattern improves the structure of the code and it gets rid of a lot of duplication in the code.

The reason for using this design pattern is mainly due to the behaviour different options and ability to change the deletion strategy in run-time. So, in our case, there will be three different strategies: one that deletes spoiled food, one that deletes food that is not yet spoiled without changing the credits of the users and one that deletes food that is not yet spoiled and subtracts the credits from the current user. This makes the code a lot more flexible. The use of the strategy design pattern reduces the amount of code

in the class that uses it by outsourcing the strategies for removing food from the database to different classes. Furthermore, the delete strategy which will be used to handle each request is decided at run-time when the optimal strategy can be selected.

How we applied this design pattern

The place where we applied this design pattern is in deleting food from the database. This can be done in one of three ways: if the food is spoiled the credits for the remaining credits are split over all users; another option is that a food may be added by mistake and so after deleting the credits are subtracted only from the user who added it; and the last option is to delete a food without subtracting any credits.
When a call is being made to the food deletion endpoint, besides the parameters that describe which food should be deleted and a token to verify that the user is authenticated, a string parameter is passed to indicate the desired deletion strategy. This makes it very easy to add more deletion strategies in the future, if such are required, resulting in a more scalable application.

As a result of using this design pattern the endpoint for deleting food works as follows: First, it authenticates the user; Then, it creates an object of class Deleter which takes as an argument the deleting strategy which should be used; Finally, the delete method of the Deleter object is called, which in return calls the delete method of the deleteStrategy.

The delete strategy is an interface and it contains only one method which has to be overwritten by any class that implements the interface. Moreover, in each implementation of this interface a different implementation of the actual food deletion is provided. Below is a diagram of the strategy design pattern.

Class diagram of the design pattern