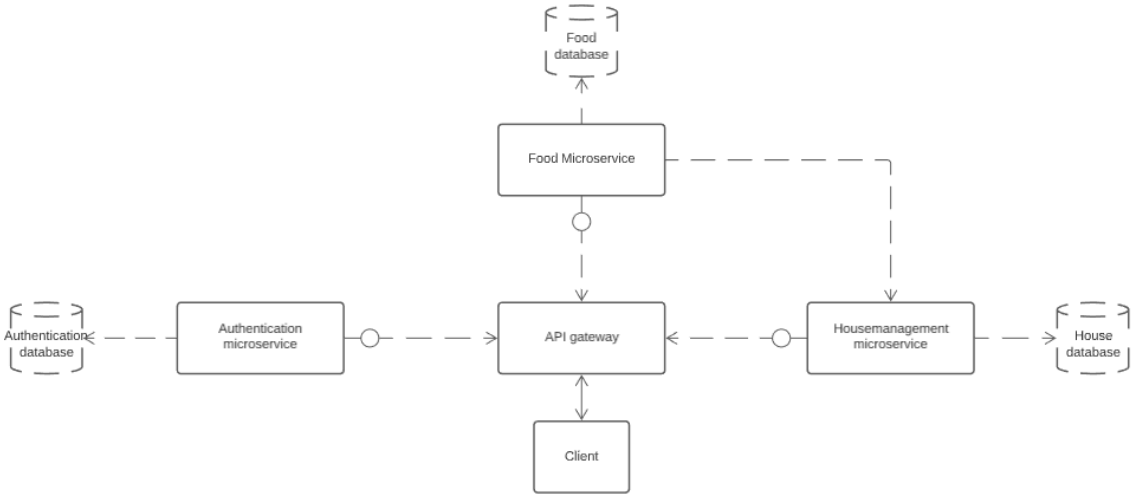
Assignment 1 - Architecture
and Design Patterns

Task 1: Software Architecture

Architecture Diagram



Overview of the design

Our application is made up of three microservices and an API Gateway.
These microservices are:

- Authentication microservice, in charge of user login & registration
- Food microservice, in charge of storage, food products and credits
- House microservice, in charge of student houses and their occupants

All user requests are sent to the API Gateway, which forwards the requests to the appropriate microservice.

Authentication microservice

This microservice is the first microservice a user will encounter, when interacting with the application, before gaining access to the rest of the application. It enables a user to register, and, if the respective account exists, to login.

One of it’s main responsibilities is to make sure a user is authenticated properly, the first time they use the application, making sure all registration parameters are valid and according to a certain standard, such as minimum password length.
It provides the API gateway with an assurance that the user is valid and may proceed.

House microservice

The house microservice is responsible for all functionality regarding the student house a user is in.
It includes the functionality to create and delete student houses, and to reset the student houses by deleting all.
A user has the ability to join or leave any student house and add a housemate to their house.

This microservice also provides functionality meant to be used only by other microservices and not by the user. This functionality includes getting the housemates of a certain user, checking if a list of users are actually housemates, and registering a user so it can join a student house.

Food microservice

The food microservice is the biggest of the three microservices, and includes all functionality regarding food and credits.
When a new user is created, food management creates a credit account for them, in which the amount of credits a user has is stored and deletes the credit account when the user gets deleted.

Any user can view their own credits, add food products, get the storage of their house, and take portions of a certain food product.
When a user has accidentally taken too many portions, they can also put back portions of the food product.
Users can also take food together, by also providing a list of users in the house with whom they want to take the portion.
The user who added a certain product can also update it and change the name, expiration date, amount, portion size and price.
Foods that have expired will automatically get deleted when a user views the storage of their house, and the leftover credits will be split over all users in the house.

Additionally to this, the microservice contains reset options to either delete all the food products in a storage, delete all food products in the database or to set the credits of all users to 0.

Interaction within the application

Our application works with Synchronous Communication using an API gateway, i.e. a microservice communicates with the API gateway, using HTTP protocol and vice versa. Hereby if one component is requesting an action it waits, until the other responses. This makes the API gateway our central connection point, enabling communication with other components.
The API gateway takes the requests of a user and forwards them to the relevant Microservice. It then takes the responses of the microservices and forwards those to the client.
The Gateway also keeps information about each c user's session, meaning we are saving time, since there is no need to keep requesting parameters, for instance the userID and the houseID, while also keeping a user logged in. If necessary, the Gateway also updates the information in the session, for example if the user leaves his house the houseID in the session is removed.
By using an API gateway and Microservices, we additionally ensure scalability, since all microservices can exist and be altered independent from each other, as mentioned above.

Why we chose this design

We have chosen the microservices in such a way, that all functionalities which have close or frequent interaction are within the same microservice. By doing this, the microservices also provide more independent functionalities. This is beneficial, since it cuts down redundant communication between microservices, and if one of the microservices were to break, the others could still remain functional.

There is also room for scalability in the application or for adding different features more easily, because the microservices are mainly split on functionality.

We have chosen to have a seperate microservice for house management, because the microservice is modularly independent of the rest of the application and the services it provides can be extended and even integrated into other applications that need it.

It is also important to have an independent microservice for authentication, separating the access points of the user credentials, in order to assure more security. If one other part of the application is attacked, the user information is not affected by this.

We have also chosen to use an API Gateway, as there are many benefits to having one.
The API Gateway saves user info after login, which prevents the user from having to send his credentials with each following request to the application, making it faster.

The Gateway also makes the application scalable, because a new microservice can be added easily, without having to change already existing microservices.
Lastly, the API Gateway improves user friendliness, since the users don’t need to know the addresses of every microservice, but just that of the Gateway, and communicate only with it since it forwards their requests to the appropriate parts of the application.

Task 2: Design Patterns:

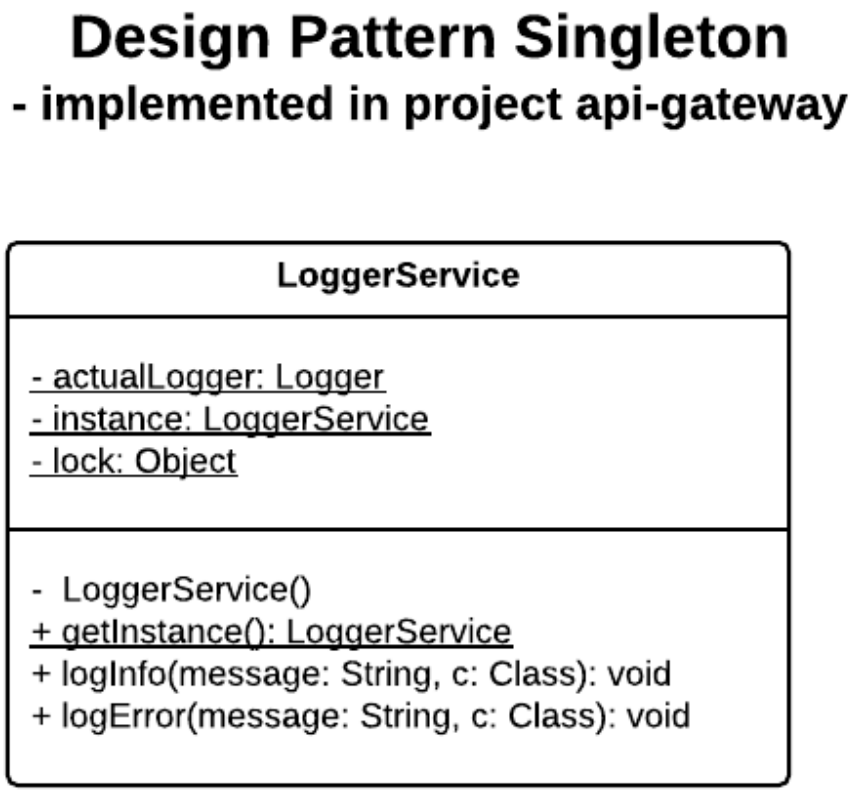
Design Pattern 1: Singleton

- Natural Language Description:

  For our API Gateway Microservice we wanted to ensure that only one instance of a logger exists at the same time, as is common for logging in applications. The reason for this is that before, we created a new logger for every class in our API Gateway microservice. This seemed impractical since nearly all classes perform logging at runtime. So, while implementing the Singleton pattern, we created a single class with one logger that can be used by all classes in the microservice.

  We applied the Singleton design pattern to the logger and implemented it by creating a LoggerService class which provides all classes in the API Gateway Microservice with logging functionalities. It contains a private constructor and the getInstance() static method as the Singleton design pattern implies. Because of this, only one instance of the logger can exist and there is a global access point, namely the getInstance() static method. Furthermore, the LoggerService class contains two methods for logging info and errors. Now that we have these two methods we can utilize the LoggerService Singleton class in the other API Gateway classes. We replaced all old logger calls with the newly created methods from LoggerService and by doing so, we have implemented the Singleton design in the API Gateway microservice.

- Class Diagram:



Design Pattern 2: Chain of responsibility

Natural Language description:

When a request for registration is received, there are multiple steps through which the Authentication microservice validates the new client credentials to ensure correctness. It has to check individual things like lack of an existing user with the provided email or username or whether the email and password conform to a certain predefined format. In addition, we expect that in the future more such checks may be added to improve the quality of the product. Because of this, we decided to utilise the Chain-of-responsibility design pattern when implementing the validation procedure.
We have applied the **Chain-of-responsibility** design pattern for the registration method in the Authentication microservice. This design pattern is called by the *registerNewUser()* method in the *UserAuthenticationService* class. It is implemented creating an interface called *Registrator*. The abstract class *BaseRegistrator* implements it, and is extended by three classes which handle the registration procedure. The classes implement the method *handle()*, which takes as parameters the *User* to be registered, the *UserRepository* and the id, which will be changed to the actual id of the new user in the database. The first class, *ValidateCredentials*, verifies the validity of the email, username and password, in order to adhere to strict safety precautions, like special characters in the password and an appropriate length. In the case that the user has inputted invalid credentials, this class will throw an *InvalidInputException*, and control is immediately handed over to the original method, which deals with the exception accordingly by returning a descriptive error response. When the credentials are verified to be valid, the handle method of the

CheckConflicts class is called. This class checks the database for any entries which have an equal username or email address. In the case of any conflicts, a *UserConflictException* is thrown. When the previous two classes have not thrown any exception, finally the *RegisterUser*'s *handle()* method is called, which finally registers the user, but throws a *DatabaseException* when it is not able to save the user into the database. After this point, control is back to the original *registerNewUser()* method, which returns a *ResponseEntity* with an appropriate message which states the result of the entire registration operation.

Class Diagram:

Please note the orange parts are just for clarification and are not part of UML, this illustrates what the calls are and how the chain is called, how control is handed over by each class and what will be returned to the original method which called the chain.