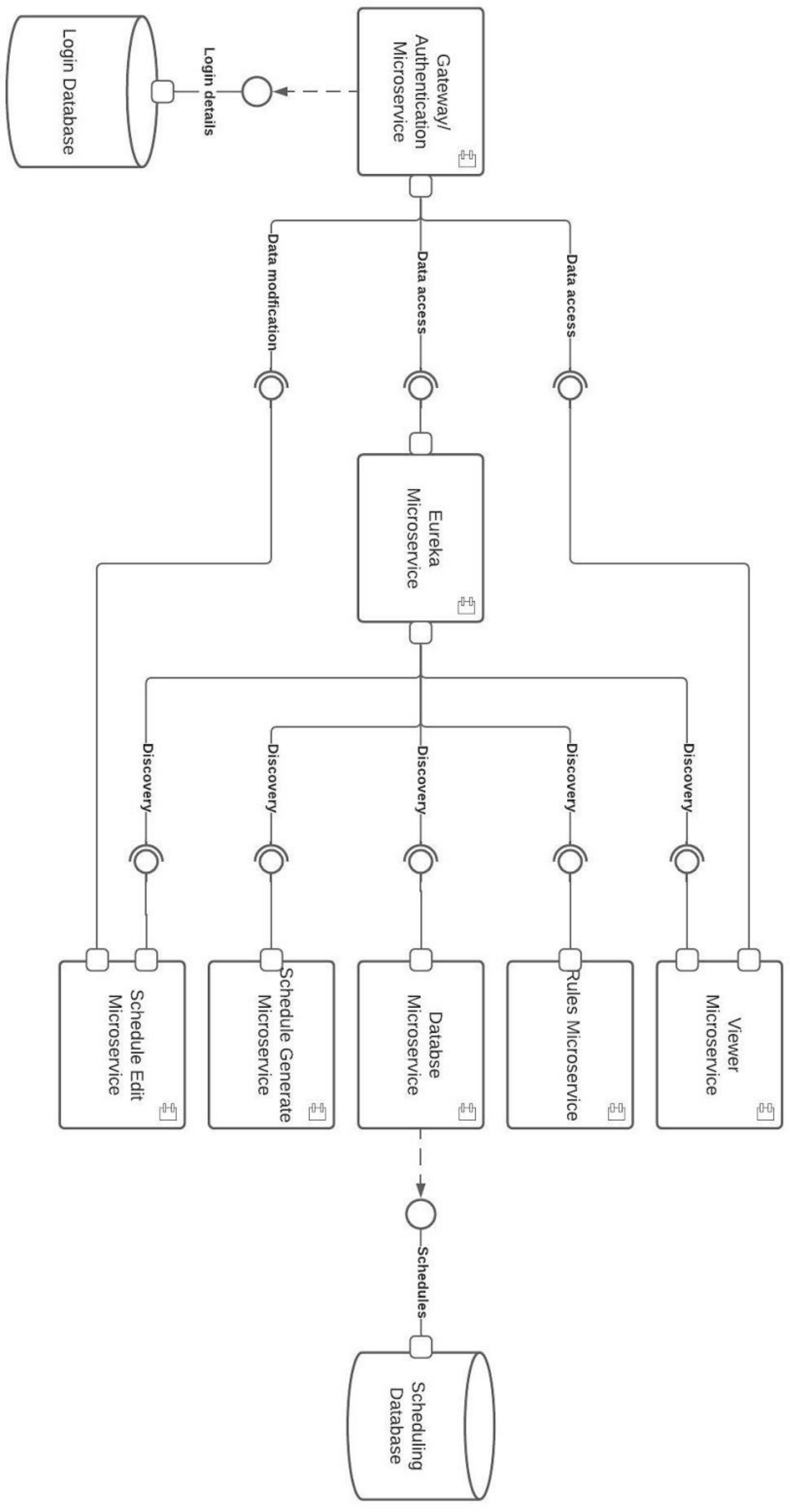
Task 1: Software architecture

UML Component Diagram



Explain whole systems architecture:

Our microservice architecture has two important microservices that define the flow of the requests, the Eureka server, and the Gateway-authentication server. Our Gateway-authentication server authenticates the users and forwards the request to the intended microservice. Eureka server simply allows the microservices to talk to each other. Upon this architecture, we created a microservice system that can communicate with itself, is accessible by one port (though the gateway), and whose requests are load-balanced.

Gateway / Authentication:

The main objective of this module is to redirect the incoming request to the intended microservice. This is achieved by a Zuul proxy server. Zuul is an edge service that proxies request to multiple backing services and provides a unified front door to our system. The major benefit of using a gateway is that it encapsulates the internal structure of the whole application. The getaway-authentication microservice is also responsible for authenticating the users. This is done by verifying whether the login details provided by the user match a record in the database. This record consists of the user netId, the encrypted password, and the type of user. Gateway-authentication microservice benefits from using tokens, allowing some part of the information to be stored on the client-side. Our service issues JWT tokens to handle authentication. The user info is stored in the token so that when the user makes a request with the token, the system knows what type of user it is. This information is used to categorize the privileges of the user, specified by the rules defined. Different privileges can access different functionalities. For example, there is some functionality embedded in our system just for teachers/admins so a student won't be permitted to use that functionality.

Eureka-server Microservice:

In our microservice architecture, some microservices depend on other microservices to be functional. For example, we decided to use one microservice to make the request to our schedule database so that we could control the requests made to that specific database. This means that when a microservice needs to make a request to the database, it needs to make a request to the database module first. We needed a way for microservices to communicate with each other in a way that didn't involve the gateway. Simply we needed to connect our microservices internally and for this job, we chose to use a eureka server which is a service discovery and registration server. All the microservices are registered to the eureka server so if a microservice needs to make a request to another microservice, it can discover the searched microservice through the eureka server.

Rules Microservice:

The Rules microservice is responsible for verifying the legality of different actions made by the schedule-generate and the schedule-edit microservices during the generation and edit of the schedules. These actions consist of applying a change to a classroom, course, or lecture. Addition of a student to a lecture, checking for the regulation classroom capacity, and checking whether each student has a chance to attend a physical lecture in the next two weeks. The specified rules are retrieved from a JSON file, allowing for quick and simple modification. In case of a change, the Rules microservice is also responsible for verifying the legality of all the existing schedules with the new ruleset. The role of this microservice is to deny any changes or insertions of the schedules that would make it not compliant with the specified ruleset.

Database Microservice:

The Database microservice is responsible for storing all the information required for generating and modifying a schedule. From the microservices responsible for creating the schedules, the Database microservice is the only one that has direct access to the database. Any modification, insertion, or deletion request is forwarded and directly executed by this microservice. It is also only this microservice that can directly retrieve any kind of data from the database. Therefore, its other role is efficient querying of any kind of information from the database. The role of this microservice in the overall infrastructure is the execution of requests to the database sent by other microservices.

Schedule-edit Microservice:

The Schedule-edit microservice has multiple responsibilities, such as adding new courses, including how many classes per week are needed, or adding newly available classrooms to the schedule. Schedule-edit checks the user type for specific operations such as adding a course. This is due to the fact that such an operation should only be allowed if the user is a faculty-member. This microservice is also responsible for making the necessary changes in case that a teacher or a student caught corona-virus. This would include removing such students from the physical lecture or moving the lecture online in case of an infected teacher.

Schedule-generate Microservice:

The Schedule-generate microservice is responsible for calculating where the new lecture should be put in the schedule, whenever an addition is made. Similarly, for each deletion operation, this microservice refactors the schedule. Operations meant by refactoring are picking a new time for a lecture, adding teachers and students to newly added lectures, etc. The Schedule-generate microservice also checks with the Rules microservice whether the database is still consistent with the regulation after the changes. If not, a refactoring in the database will not be made. If a change was attempted to be made too close to an event time (48 hours prior), the microservice will not perform any rescheduling, since this would give too little time to students and teachers.

Viewer Microservice:

The Viewer microservice is responsible for handling requests made by users. These requests are predominantly requests made by students and teachers asking for their schedules. Allowing them to see the lectures, as well as the locations, for which the students and teachers are scheduled.

Why did the development team choose the presented structure?

Using microservice architecture means dividing the whole system into flexible, maintainable, robust, and scalable pieces. Microservice architecture increases productivity and speed for distributed teams working simultaneously. The development would be much more confusing and complex in a monolith system. Another benefit of this architecture is scalability if you expect the product to grow in the near future. Our architecture uses two databases, the Login database, and the Scheduling database. Login database is directly accessed by the authentication microservice and contains login information. While the Scheduling database contains the scheduling information, its division would be difficult and potentially cause more issues than benefits. We decided not to split the data across more databases since we believe that further division of data would be unbeneficial for our system, providing little to no speed up and causing unnecessary consistency liability. Our architecture benefits from a Eureka server (also known as Discovery server). Which is a server used in microservice architecture that contains the information about other microservices. Every microservice is connected to Eureka, in turn, the Eureka server memorizes the services running on each port. Usage of the Eureka server allows the system to have a much simpler architecture with easier maintenance compared to connecting everything directly to each-other. It is only the viewer and schedule-edit microservices that are directly connected to the authentication/ gateway microservice, as these are the only services that should be viewed (and edited) directly by the user. The other microservices are encapsulated in a way where they are connected internally while remaining invisible to the user.

Task 2: Design patterns

2.1 Observer Pattern

One of the additional features we had was that users should receive a notification whenever a change in their schedule ensued. Since we had not added this feature yet, we decided to implement it using the observer design pattern. *"This is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing."(Refactoring.Guru, 2020)* Using this design pattern we can notify users of any schedule changes.

In our codebase, we identify a subscriber and a subject. The subject is the lecture schema in the database. Subscribers are students and teachers that are notified when a change to their schedule happens. Following the observer pattern, we have implemented a mechanism such that objects can subscribe to a stream of events from a publisher. We do not need to create a class that interacts directly with the database, because spring already handles this, so we have created the LecturePublisher class. This LecturePublisher class will be autowired into any component that wants to add/remove subscribers and in any object that interacts with the lecture schema. When a component wants to register a new subscriber it will call the add subscriber method with the corresponding UserNotifier. When there is an update to the database, the class responsible for it will now call *notify_subscribers*, which causes the LecturePublisher to go over all the subscribed LectureSubscribers and call the update method. The subscriber will then determine what to do with the update, depending on its type. We have decided to create an abstract class (UserNotifier) that implements the LectureSubscriber interface and automatically decides the reason the update method was called (using our enumerated "actions" and "actors") and it will call the corresponding abstract method which is defined in TeacherNotifier and StudentNotifier. A good example is that when a teacher (actor=teacher) reports that he's sick, the lecture is going to be moved to be online (action=lecture moved online), and therefore only students will be updated. The decision to set UserNotifier to be abstract is in order to force implementations to be user-type-specific, in order to make it easier in the long-term to define custom behaviors. Finally, these notifiers shall use the appropriate API call to notify the students.

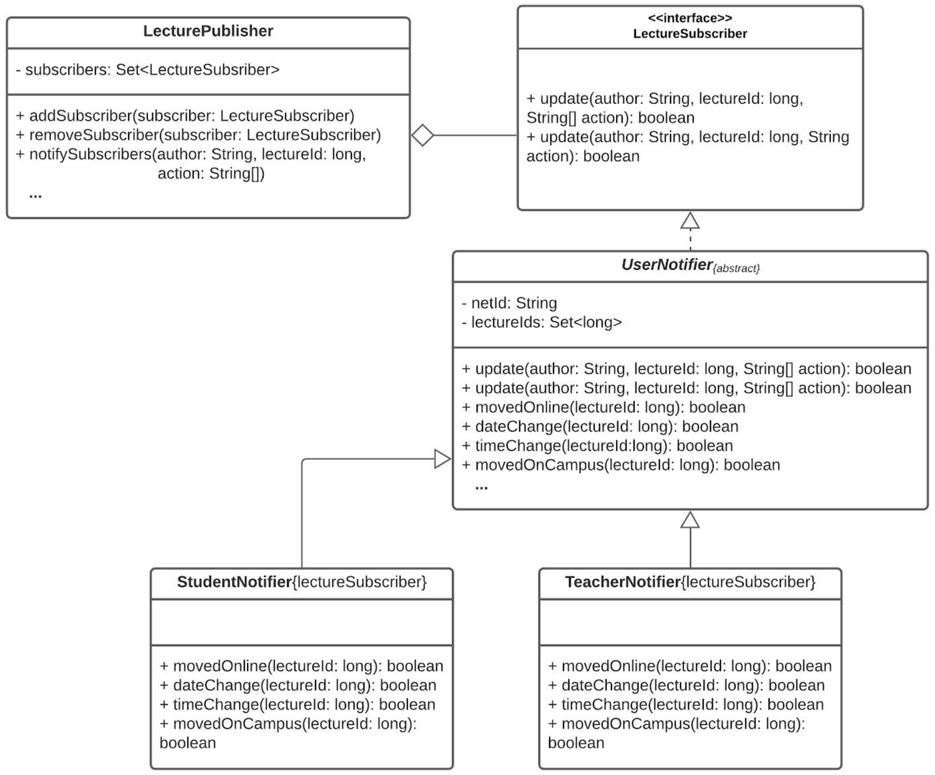

LecturePublisher
- subscribers: Set<LectureSubsriber>
+ addSubscriber(subscriber: LectureSubscriber)
+ removeSubscriber(subscriber: LectureSubscriber)
+ notifySubscribers(author: String, lectureId: long, action: String[])
...
<<interface>>
LectureSubscriber
+ update(author: String, lectureId: long, String[] action): boolean
+ update(author: String, lectureId: long, String action): boolean
UserNotifier{abstract}
- netId: String
- lectureIds: Set<long>
+ update(author: String, lectureId: long, String[] action): boolean
+ update(author: String, lectureId: long, String[] action): boolean
+ movedOnline(lectureId: long): boolean
+ dateChange(lectureId: long): boolean
+ timeChange(lectureId:long): boolean
+ movedOnCampus(lectureId: long): boolean
...
StudentNotifier{lectureSubscriber}
+ movedOnline(lectureId: long): boolean
+ dateChange(lectureId: long): boolean
+ timeChange(lectureId: long): boolean
+ movedOnCampus(lectureId: long): boolean
TeacherNotifier{lectureSubscriber}
+ movedOnline(lectureId: long): boolean
+ dateChange(lectureId: long): boolean
+ timeChange(lectureId: long): boolean
+ movedOnCampus(lectureId: long): boolean

Figure 1: The UML diagram of the observer pattern.

2.2 Builder Pattern

We decided to implement a builder pattern in the gateway authentication module to streamline the process of parsing arguments and forwarding API calls to the other modules. Before this pattern was implemented, the SysInteractController class, which is responsible for taking incoming API calls and redirecting them to the appropriate microservice, receives arguments stored in a list of undefined objects, and it was assumed that the order in which these objects had to arrive was known by both the sender (the service making use of our API) and the receiver (the gateway authentication module). This limited design space since whenever a new feature is added, the specific order of arguments would need to be communicated to all of our clients, greatly slowing down both the design and production process of future features.

The builder pattern solves this issue by creating an arguments class, which can store all of the relevant arguments for a request in specific attributes, and an argument builder (ArgsBuilder) class, which can create an argument object and, using the arguments given by the sender, fill this object with the information necessary for the SysInteractController to then forward onto the other modules. The input arguments given by the sender are stored in a Dictionary, argsDict, as you can see in fig. 2. This makes accessing the user input much more reliable, as the system no longer relies on the client's knowledge of how their input needs to be parsed. As shown in fig. 2, the ArgumentsBuilder class is an implementation of the Builder interface, which allows for similar builders to be created if the range of possible API calls, and their accompanying arguments, necessitate it.

Several methods of the argument builder retrieve multiple arguments from the input at once, which greatly reduces the amount of code used in the SysInteractController class involving more complicated data types such as adding a lecture or course. This is ideal as the primary function of the SysInteractController is to relay incoming messages to the modules who need to receive them, and so by reducing the amount of code in its methods, each method of the class has a much more well-defined purpose and is easier for collaborators to read and understand the purpose of.

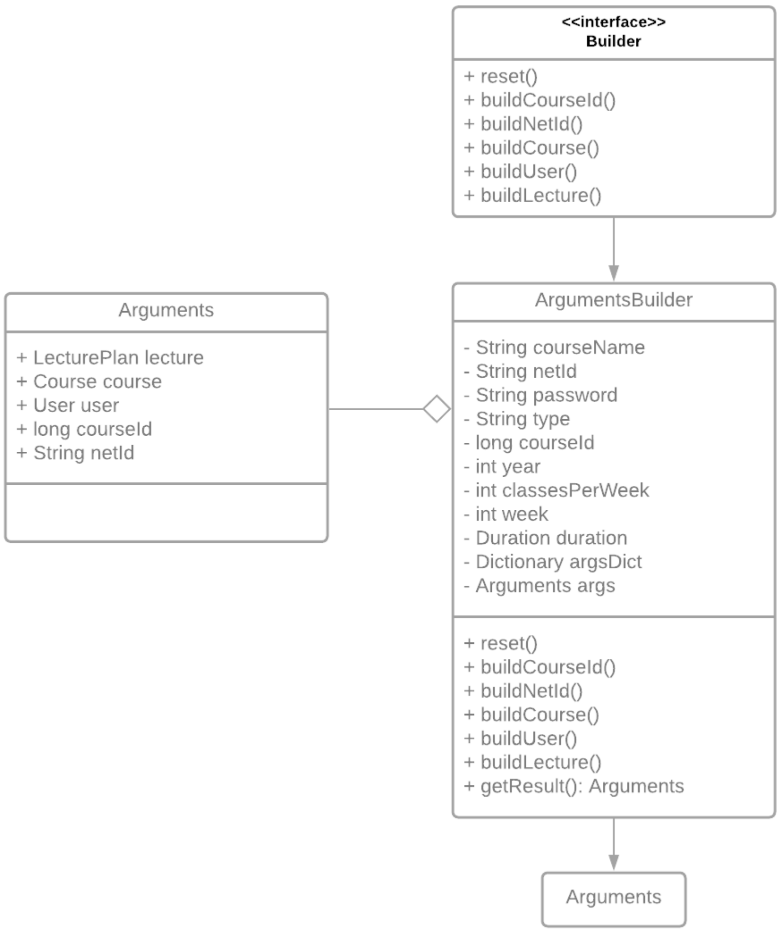


Figure 2: The UML diagram of the builder pattern

2.3 Proxy Pattern

Before receiving this assignment, we implemented a proxy pattern through the use of a dedicated database microservice, to which all other microservices sent database queries, instead of the database itself. By having all interactions with the database be handled in one place, we can easily create and enforce rules about how the database is accessed and what operations are allowed to be performed; by upholding these rules in the database module, it is upheld in the entire system as well.

Using a proxy to control access to the database also makes it simpler to communicate with the database, as developers working on other microservices do not need to understand the schema of the database in its entirety, they just need to know what call to make to the “Proxy server”.

Sources:

Refactoring.Guru. (2020, January 1). Observer. https://refactoring.guru/design-patterns/observer