Software Engineering Methods

CSE2115

Assignment 1

Phases of Project Management
The Marketing Funnel
Awareness
Interest
Projected Budget
Project Close
Project Performance and Control
Project Definition and Planning
Project Conception and Initiation
Project Launch or Execution

Source: https://www.pexels.com/nl-nl/foto/man-mensen-vrouw-bureau-7605981/

Group 29b

Group members:
Natanael Djajadi
Elena Mihalache
Jordan Sassoon
Paul Tervoort
Yunhan Wang
Jocelyn Woods

Task 1: Software architecture

1.1 Bounded contexts

Using domain-driven design we have identified the following bounded contexts: authentication, user, reservation, facility and lesson. These will be explained one by one, arguing upon the choice to make each of them a subdomain of their own.

Authentication Context: This subdomain concerns mostly the validation of users. It decides whether the user, who provided the username and a password, is valid, by comparing the provided login credentials to the data saved in the database. If the user is valid, then it can notify the other contexts without those having to consider authentication. Confirming that a user is valid is important in our architecture, because otherwise users can reserve for other users and fill their daily reservation limit. More unpleasant is if a person reserves some piece of equipment, then destroys or loses it. This individual could then access the system easily without having admin rights, and set the recent user of that equipment to NULL or nobody. With this subdomain, other contexts can be more assured that the user is the person they claim to be, than they would be without authentication. Lastly, the password will be stored safely, with hashing. So people who break into the system cannot easily get the passwords.

User Context: This context is a subdomain that contains both the functionalities of user and team, since each user needs to be in a team in order to reserve. If a single user is reserving then they are counted as a team of one. One team can contain many users and one user can be in many teams, so there is a many to many relationship between those two entities. We decided to include team in the user context for easing the interaction between the two. The reservation context (see explanation below) may want to check how many users are in the team to decide whether the reservation is accepted for the team. Moreover, the reservation context may want to see the role of the user (basic or premium) to determine if the team can reserve without exceeding the daily maximum amount of reservations of each team member.

Reservation Context: This context is the main subspace of the project. It offers functionalities for reserving both facilities and equipment. It also stores the relevant information in the event of a user signing up for lessons. Every user interaction with the facility is saved and retrievable with the use of timestamps. This bounded context also contains the logic for halls, sports and equipment management. The reservation context is therefore highly dependent on the surrounding microservices, as a single reservation requires a team identifier, an optional lesson identifier and a team size.

Facility Context: The facility context is a subdomain which contains functionalities for the sports halls and fields, but also for the diverse types of sports and equipment they require. Each sports hall or field can be used by at least one sport and each sport can have one or more types of equipment needed or not. The decision that sport and equipment are to be included in this specific context is argued by the dependency they have on each other. The facility also has a relationship with reservation, as the former requires information about the facility name and equipment requested.

Lesson Context: This context offers functionalities for creating and deleting lessons. It is dependent on the hall and the sport, as it requires the names of both the hall where it takes place and the sport it is about. It does not concern the user, authentication, or reservations. That is the main reason it is a subdomain on its own.

1.2 Microservices

The bounded contexts have to be mapped into microservices, since this is the chosen architecture. The API Gateway infrastructure has also been added to the design. The services will now be explained one by one, detailing their role within the architecture, how they interact with one another and the rationale of the choices made.

Authentication service handles the authentication of user entities within the system. When the user gives a username and a password, it returns a boolean regarding the user's validity. The user is valid if the username and password match a username and password in the database. If the user is valid, then it can forward them to the User service. This service then knows that the user is authenticated, and can give permissions to the user to do things like creating a team, getting information such as the users registered in a team, and retrieving the role of a user. If the user is not valid, then they will not get permission to access other features of the software. If the user were to get permission with only a username and no password, then they can pretend to be another person and can make reservations for them.

User service handles the storage and retrieval of user data and team data such as attributes like username, team name, and the size of a team. It also handles the modifications of user entities and team entities if they're correctly authenticated by the Authentication service. The data is stored in the database of the User service. This service can return the role of a particular user, the team ids of a specific user, as well as the size of a team, used for the Authentication service and the Reservation service. The User service has a controller that checks if there are any incoming requests with correct endpoints and processes these requests. The request can be storing a new user, retrieving the number of users registered in a team, and so forth. With a correct join token, the user can be inserted into an existing team. Team is included in the User service at the convenience of joining them. This can also reduce the overall complexity of the system and the overhead of network communication.

Reservation service deals with creating and modifying reservations, and getting information about them. Only admins or members of the group for reservation can view and modify it. Any type of user can make reservations, provided that the user has made less reservations than the maximum allowed reservations for the user's type. To check this, the requesting user is first authenticated by the authentication service, and then the user type is requested from the User service. The reservation can be made for a group or a single user. If a user makes a request to create a reservation for himself, the reservation service will under the hood use a group with only one person. This is done, because it makes the implementation easier than that groups as well as individual users will be saved into a reservation. Then a reservation needs both a team ID and a user ID in their class, but if you only have teams then only one of the two will be saved, which eases the implementation. If a lesson is reserved, the lesson service is requested to check if the provided lesson exists, and the capacity of the lesson. Before making the reservation, the service queries its database to count the number of reservations, to check if the reservation fits in the capacity.

Facility service handles the creation of sports and equipment provided by the centre and the manipulation of halls or fields and of the aforementioned by storing all the details in its own database. This service can return the sports which can be played and the equipment available for each location, information used by the Lesson (see explanation below) and Reservation services. The controller checks the endpoints and responds to requests. Like earlier discussed, sport and equipment are both in one domain since they have a dependency on each other. This will be also reflected in this microservice, by having those two classes in one service.

Lesson Service handles the creation and manipulation of lessons and retrieving information about them. The information about lessons is stored in its own database. To do modifying or creation of lessons, a user must have admin privilege. If users' privilege cannot be verified, then all users will be allowed to create lessons. This can be disturbing, mostly in situations where a real teacher wants to schedule a lesson and a random user also creates a lesson at the same time and in the same location, then the official lesson cannot be scheduled. The requested user is therefore authenticated in the authentication service and the role is requested by the User service. If a user is an admin, he/she can create lessons, update their properties and remove lessons. Any user can view the properties of a lesson and search for lessons using some possible constraints or filters that are provided in the API.

API Gateway serves as a central interface between a client and backend services. Here, the client can also be another microservice. The gateway receives all API calls and routes them to the appropriate microservices. Its convenience comes from being a single entry point for all requests, handling the connections between services. This removes the distribution of connection information across the application, rendering microservices unknown to each other. To store the network information of the services the API gateway uses a service registry, mapping request URLs to the correct microservice ports. The gateway acts as a medium for information routing, enabling easier microservice interactions and a more secure design.

By adopting a microservice architecture, the business logic is separated into bounded contexts, which makes the code in each microservice clear and easy to understand and to maintain. Moreover, different microservices can be monitored and extended separately, and requests are sent only to the microservices that are supposed to handle them, reducing the load on a single service. Finally, development can be distributed easily: different teams can work independently on their own microservice.

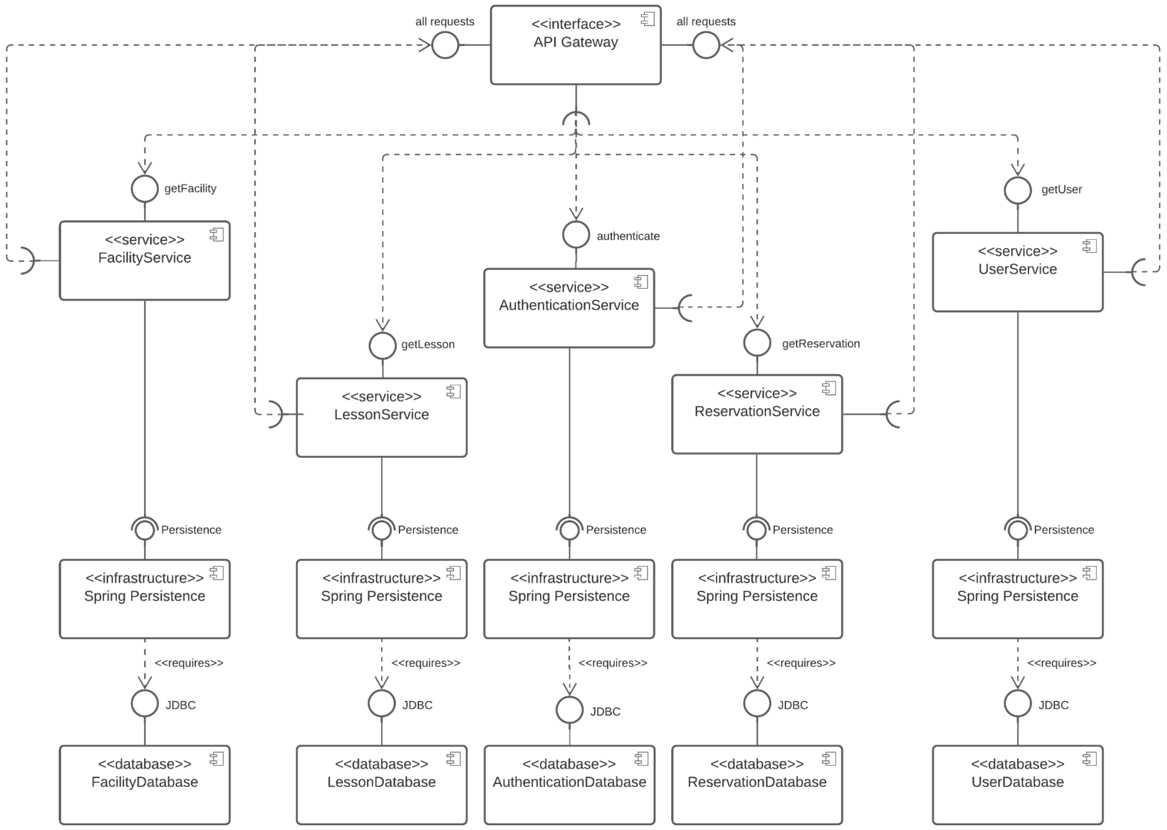
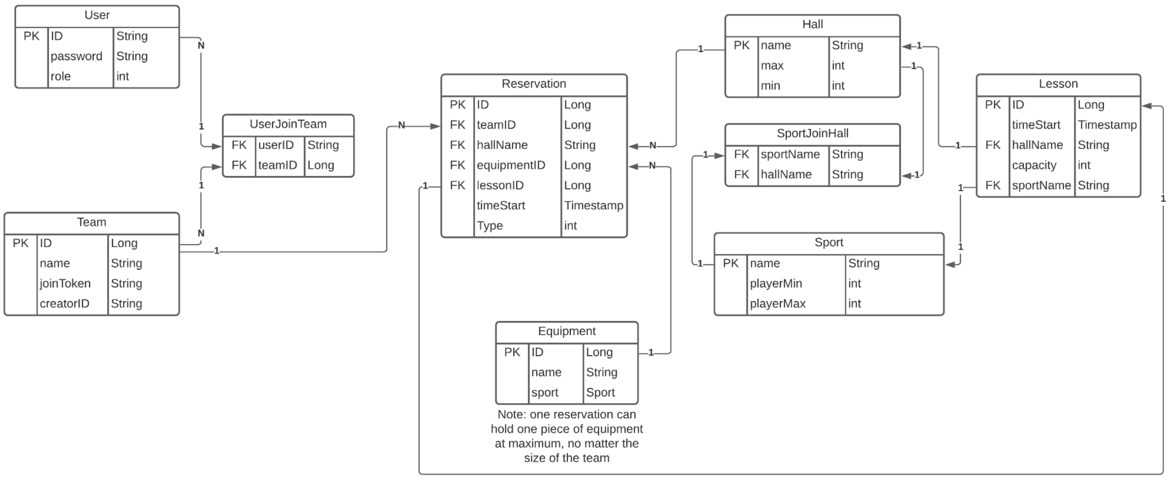


Figure 1 - Architecture diagram



Note: one reservation can hold one piece of equipment at maximum, no matter the size of the team

Figure 2 - Database schema

Task 2: Design patterns

2.1 Builder pattern:

1. Pattern description:

   The Builder design pattern separates the construction of an object from its representation and can be used to create different representations of the object. This is done with a step-by-step process to create the product. One of the reasons for this is the independence of the client. Only the builder knows the specifics. It is best to use this pattern when a complex object's parts are assembled independently, and the construction allows different representations of the object. The overall consequences of using this design pattern are varying the object's internal representation, isolating the code relation to the construction and representation of the object, and giving more control over the construction process.

   The Builder design pattern is part of the creational branch along with the Factory Method, the Abstract Factory, and Singleton. To better understand why the Builder pattern was chosen, a comparison between them might be helpful. The Factory Method is more focused on the instantiation of different objects through the same API, while the Abstract Factory is best at emphasising the families of the product objects (simple or complex), when creating them. In contrast to these, the Builder focuses on the construction of each complex object in a step-by-step manner.

   We have implemented the builder pattern for the creation of abstract 'Location' objects. A location can either be a 'Hall' or a 'Field', and both types have a name, minimal group size, maximal group size, and a set of sports that can be done at that location. The builder can then build the object using those parameters. The concrete builder can be a HallBuilder or a FieldBuilder. Both builder types return a Location.

2. Class diagram

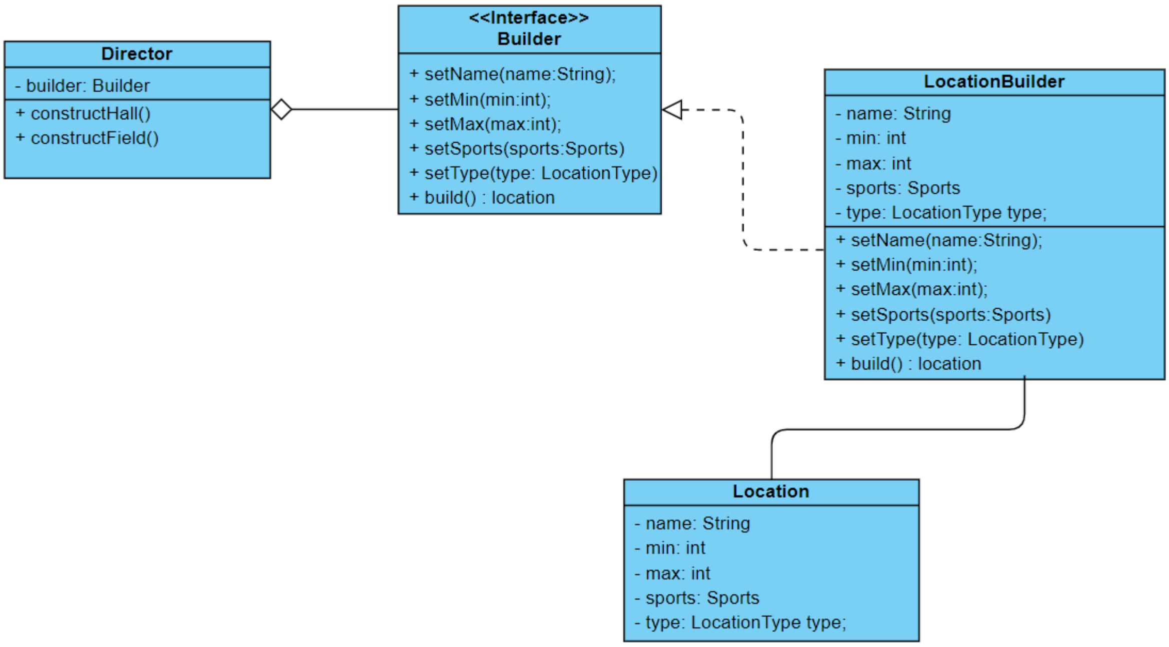


Figure 3 - Builder pattern class diagram

3. Pattern implementation

- facilityservice/src/main/java/nl/tudelft/sem/facilityservice/application/LocationController.java
- facilityservice/src/main/java/nl/tudelft/sem/facilityservice/domain/Builder.java
- facilityservice/src/main/java/nl/tudelft/sem/facilityservice/domain/Director.java
- facilityservice/src/main/java/nl/tudelft/sem/facilityservice/domain/LocationBuilder.java
- facilityservice/src/main/java/nl/tudelft/sem/facilityservice/domain/Location.java
- facilityservice/src/main/java/nl/tudelft/sem/facilityservice/domain/LocationType.java
- facilityservice/src/main/java/nl/tudelft/sem/facilityservice/domain/Sport.java

2.2 Facade pattern

1. Pattern description

The facade pattern facilitates to abstract a complex implementation into a simplified interface such that the underlying implementation is hidden and easy to use on a higher level. In the User service, User objects will be converted to UserDto objects as well as Team objects in controllers, the conversion involves a set of complex functional programming usages. Hence, it is decided to implement some utility methods to realise the conversions and provide an easy-to-use interface for the sub-system. Thus, the readability of the methods in the controllers is improved due to their simpler implementations. Also, if there are more methods that need to convert, the utility methods can be reused. If we do not adopt a facade pattern and choose to implement the conversions in the original methods, a change in the logic of the conversion requires modification of all methods that implement the conversion. It can cause some severe problems if we forget to fix a method that should have its logic fixed.

Moreover, the idea of API gateway also makes use of the facade design pattern. The API gateway in the implementation receives all the requests from clients and redirects them to microservices. This eases the cooperation between microservices since there is only a single entry point for the clients to request with.

There can be some downsides of using the facade pattern. For example, if there are many unrelated utility methods and we include all of them inside a single class, this class itself can become complex and messy to work with. Furthermore, there is more management and orchestration required to maintain the API gateway. However, it is believed that the advantages of using the facade pattern can overthrow its downsides.

2. Class diagram

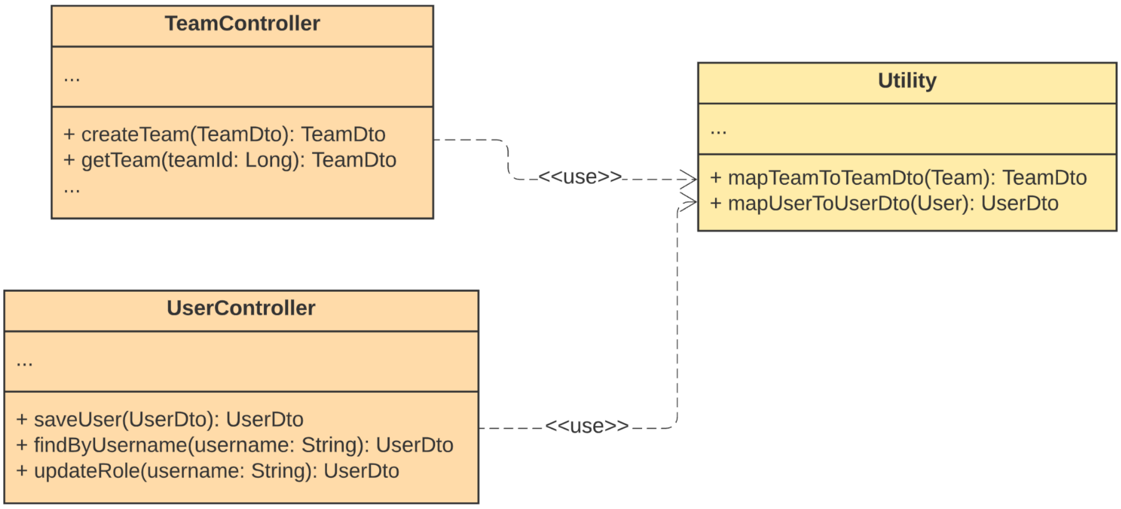

TeamController
...
+ createTeam(TeamDto): TeamDto
+ getTeam(teamId: Long): TeamDto
...
<<use>>
Utility
...
+ mapTeamToTeamDto(Team): TeamDto
+ mapUserToUserDto(User): UserDto
UserController
...
+ saveUser(UserDto): UserDto
+ findByUsername(username: String): UserDto
+ updateRole(username: String): UserDto
<<use>>

Figure 4 - Facade pattern class diagram (utility)

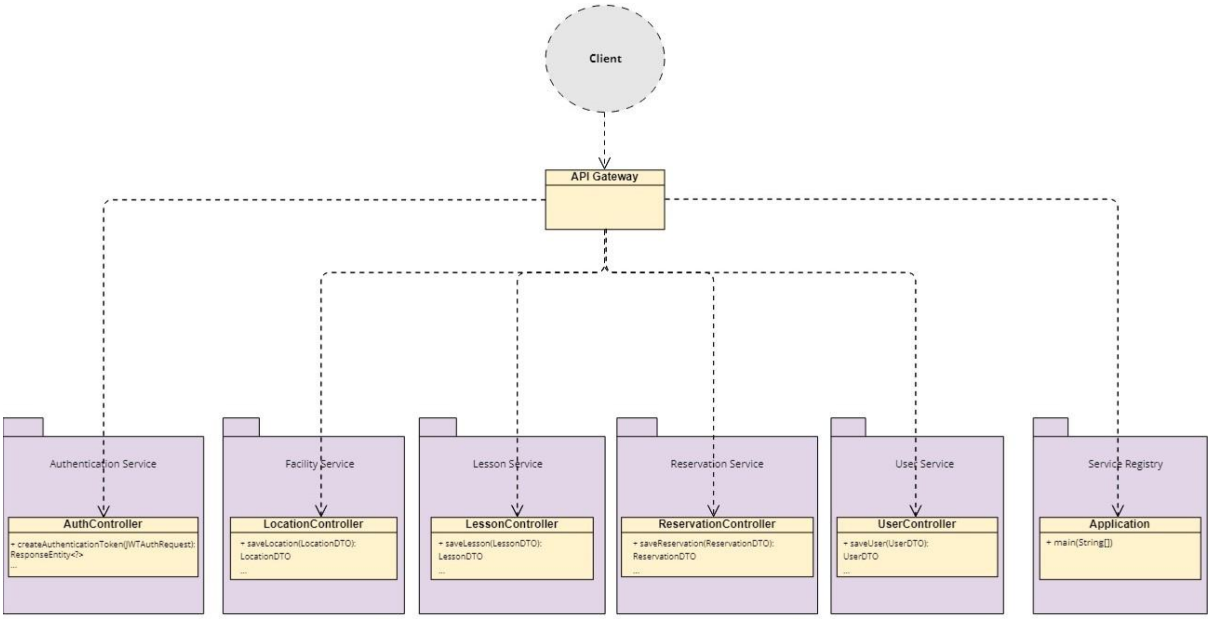


Figure 5 - Facade pattern class diagram (API gateway)

3. Pattern implementation (File paths in the project folder)

- userservice/src/main/java/nl/tudelft/sem/userservice/domain/Utility.java
- userservice/src/main/java/nl/tudelft/sem/userservice/application/TeamController.java
- userservice/src/main/java/nl/tudelft/sem/userservice/application/UserController.java
- apigateway

Bibliography:

1. Picture on the cover, available at: https://www.pexels.com/nl-nl/foto/man-mensen-vrouw-bureau-7605981/

2. stackoverflow.com, “How to show usage of static methods UML Class Diagram”, answered by RobertMS June 26, 2012, available at: https://stackoverflow.com/questions/11209240/how-to-show-usage-of-static-methods-uml-class-diagram (Accessed: 2021/12/14)