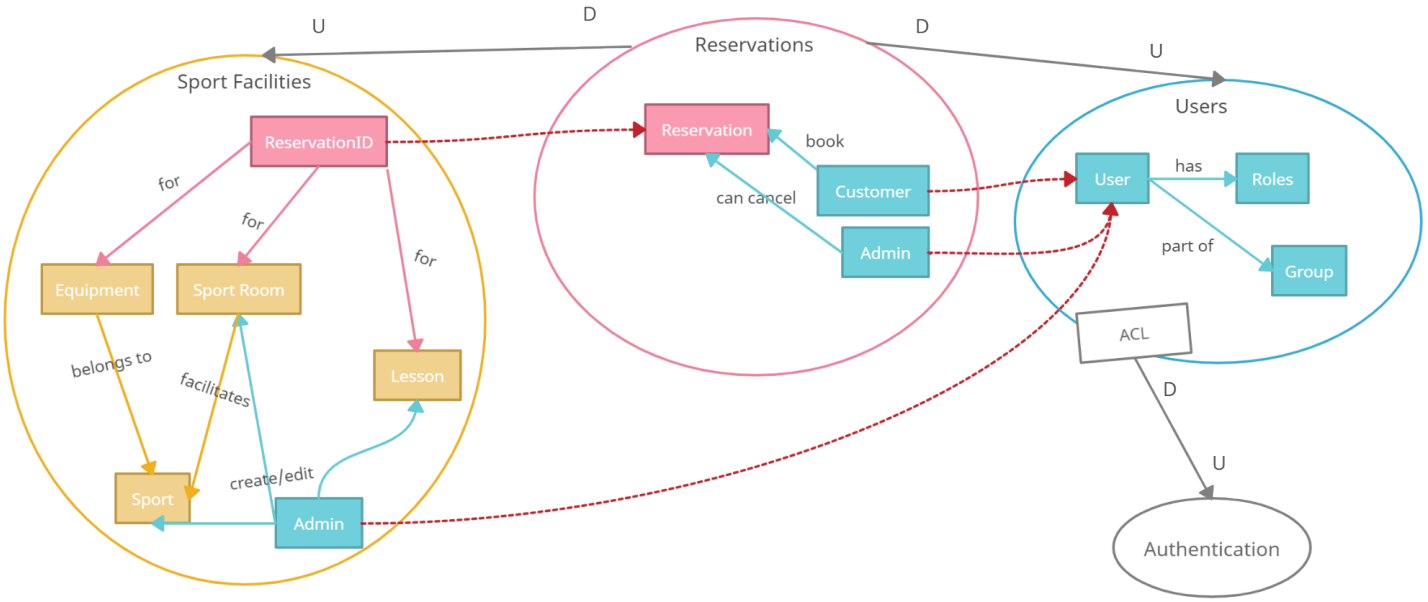
Task 1: Software architecture

For our system, we are using a *microservice architecture*. It consists of a central API gateway and our internal microservices, each having its own database.
The API gateway routes every request to the respective microservice and every microservice has its own API (called “Controllers” in our system, as we use the *Model-View-Controller architecture* within one microservice). Microservices can then communicate via these APIs to handle requests.
Below we explain how we have derived our microservices.

Bounded contexts (DDD)

After having derived the requirements and applied Domain Driven Design (DDD), given our scenario for a sports centre booking system, we ended up with the four bounded contexts that are shown in the context map below. Using the domain-related keywords in the given scenario, we were able to build the ubiquitous language.



Context map: Sports Centre Booking system sub-domains

By splitting the whole domain into sub-concepts, and thus dividing our system this way, we can ensure that our system and its code will not get overly complicated and that it can easily be maintained over time, since well-defined modules provide cohesion. However, at the same time we want to keep coupling between modules low, to facilitate future modification and minimize code duplication.

And thus, the bounded contexts that we have come up with for the given scenario for a sports centre booking system are:

- Users
- Reservations
- Sports facilities (lessons, sports, sport rooms, equipment)
- Authentication

The first three contexts (*Sport Facilities, Reservations, Users*) are our *core domains* that our system focuses on, since the application would not work the way it should work without these three. All these contexts are essential parts of the application: handling users, reservations and sport facilities.

The reason that we decided to combine the entities lesson, sport, sport room and equipment into one context named "*Sport Facilities*" is that they are all either very connected, thus this would form a cohesive sub-context. What we mean by this is that lessons, sports rooms and equipment serve a similar purpose, namely to be allocated to a customer whenever a reservation is made.

Despite the fact that our given scenario referred to "booking lessons" as a separate functionality from "reserving equipment/sport rooms", we figured these two concepts were highly cohesive, thus within our system lessons are treated as "reservations".

This is all done within the reservation context, so by combining these concepts in this way, the reservation service can always communicate with the same service when a reservation is asked to be made, and hence we keep the amount of communication between different services low. The sport entity is not similar to the latter three, however, it is inherently connected to them because lessons, sport rooms and equipment all have a sport related to them.

As can be seen by the Downstream-Upstream relation between the contexts, our Reservation context is influenced by the User and the Sports Facilities context; it requires data from them. Examples would be checking whether a user has a basic or premium subscription, or whether or not there is sufficient equipment in stock.

Authentication, however, is a *generic domain*, an additional technical domain. This is because even though it is needed for the application, it is not "core" to it. This is also something that can be outsourced, as the modelled Anti-Corruption Layer acts as a "translation" layer (Spring Security).

Mapping bounded context to microservices

When deciding how we wanted to map these domains to microservices, we thought about how related each of the domains were to the other ones. We wanted to make sure that if domains are highly coupled, they are in the same service while still ensuring all the components are cohesive enough.

We decided to decompose our application by subdomain, organizing services around DDD. Hence, all the domains, namely users, reservations, sports facilities and authentication are each mapped to their own microservice. We chose not to merge these with another domain because they are simply too distinct to be combined with a different domain, mainly since we already decided to make one context sport facilities of four entities instead of making them all a separate context. If we were to merge these domains, we would get very incohesive microservices as a result, leading to unmaintainable code and defeating the purpose of modularity provided by the chosen microservice architecture.

And thus the final microservices we have decided on are the following: *User Service, Reservation Service, Sport Facilities Service* and *Security Service*.

The *User Service* maintains all information related to the users, handles user-related features, such as registration, and provides other services with information about the user. A user can be either a customer or an admin. For customers, a unique ID, username, password, and subscription type are stored. The same attributes are stored for admins, except for the subscription type. To improve security, a user's password gets encoded before it is stored in the database.
Any person can become a customer by simply registering. In contrast, admins can only be registered by a fellow admin. For that reason, one admin is hard coded into the database.
Another property that is stored by the *User Service* is the group feature. Customers can make a group together, that consists of their peers, to make a reservation together. Like users, groups also are assigned a unique ID. Alongside this ID, group name, size and members are stored as well.
Lastly, the user service provides information about the users to the reservation- and security service. The reservation service requires, for instance, whether a user is premium, or the size of a group, while the security service needs to check if the login credentials match those in the database.

The *Reservation Service* manages reservation data, handles creating reservations, and verifies whether a reservation is valid. Validity is based on multiple variables, such as, but not limited to, the amount of equipment in stock, availability of the desired sports room, and whether the minimum group size is reached.
Reservations can be made as an individual customer, or as a group.
The facilities that can be booked are sports rooms, a piece of equipment or a spot for a lesson.

The number of reservations that can be made depend on the subscription type of the customer, a basic user is limited to one reservation per day, while a premium user can make up to three. However, only reservations for a sports room count towards the total (not equipment or lesson reservations). Otherwise, it would not be possible for a basic user to reserve a sports room and have the necessary equipment on the side, to practice the sport.
After a reservation is made, and verified as valid, it gets stored in the database. The database stores who made the reservation, what was reserved, and the corresponding starting date. Moreover, a unique ID is assigned to every reservation.
The reservation service's interactions with other services mainly consist of asking for information rather than providing it (*Downstream context*). For example, from the *User Service* it receives the user's subscription type and from the *Sport Facilities Service* it gets the capacity of a sports hall.

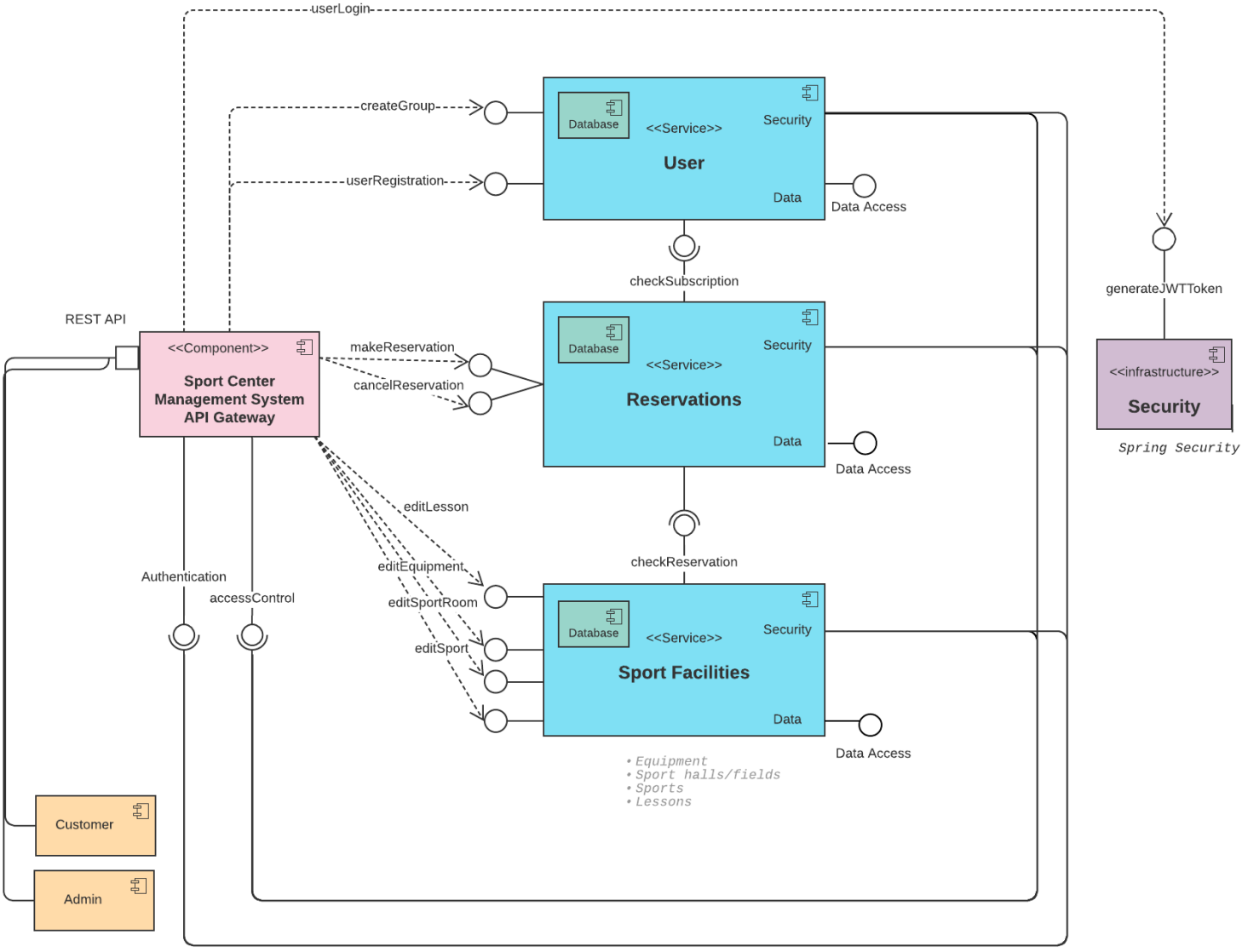
The *Sports Facilities Service* stores the equipment-, sport room-, lesson- and sports information, which is later accessed by the *Reservation Service.*
Every entity has different attributes that the *Reservation Service* could potentially need.
Firstly, for each piece of equipment the service stores a unique ID, a name that represents the type of equipment, the sport it is related to, and finally whether it is free or in use. An example of a name for a piece of equipment is "hockey stick" and the related sport would be hockey.
Secondly, for sport rooms a unique ID, the type of sport room, min/max capacity, a list of related sport(s), and the name of the sports room is stored.
There are two types of sport rooms, halls and fields. A sport field is dedicated to a single sport, whereas a variety of sports can be performed in a sports hall. Therefore, in the case of a sports field, the list of related sports merely contains 1 sport.
Thirdly, the lesson data stored consists of a unique ID, title, starting/ending time, and size. The title represents what type of lesson it is, for example "tango", and size is how many spots the lesson has.
Lastly, information about the sports provided by the sports centre is stored. The attributes stored are the name of the sport, whether it is a team sport or not, and min/max team size constraints.
The latter is used to check against the customer's group size, when a group reservation is made for a team sport. (For soccer, a system admin could decide to allow group sizes between 8 and 14 e.g.) For non-team sports, the min- and max size are set to -1 and 1 respectively.

The main purpose of this microservice is to store information about all facilities that can be reserved. Therefore, it only interacts with the reservation service, by providing it with information.

The *Security Service* handles two tasks, authentication and authorization. Authentication means verifying the identity of a user, and authorization is deciding which user has what rights. It completes these two responsibilities by issuing JWT tokens to users that have successfully logged in. JWT tokens are a token from which the system can verify the identity of a user. Therefore, a user needs to only authenticate themselves once, since the token can be passed along with every request.
Additionally, authorization information is also stored in this token, since the system can verify whether a user is a customer or an admin. The default authorization that is required for a method is for the user to be authenticated. However, some methods, such as, adding/removing equipment or registering a new admin, require the user to be an admin.
Finally, there are functionalities that need to be whitelisted to not require any form of authentication, such as logging in. The security service only interacts with the *User Service*, to see if a user's login credentials match those found in the database.

The *API gateway* serves as a single-entry point for users, by routing every request to the corresponding service. Moreover, this gateway allows us to have a central point where we can configure authentication and authorization. Before routing every request, the gateway checks whether the user is authenticated and has the access rights, by verifying the token that was issued by the *Security Service.*
This explains why our microservice component diagram below illustrates *authorization* and *access control* as functionalities provided by the API gateway.

We selected these four microservices because it achieves the simplest, and most cohesive way to fulfill all the requirements of the system, while at the same time keeping our codebase as maintainable as possible. Every microservice is essential to the system, and omitting one would make the application unusable.



UML component diagram: all microservices within our Sports Centre Booking System Application

Task 2: Design patterns

1. Strategy Pattern

Why is this pattern implemented?

The implementation of the strategy pattern in this application can be used to process the list of reservations in different ways. This is done by creating a family of algorithms, encapsulating them and making them interchangeable.
This pattern uses the interface as an alternative to subclassing; consequently, eliminating conditional statements. This is in respect to the “Open-Close Principle”, since we can extend our system with as many concrete sorting strategies as we like, without having to change production code.
An admin is given the choice of sorting the reservation list according to the demand or requirement. An example of a requirement can be sorting in ascending or descending order of username. Another useful technique for the admin is to sort the reservation list in chronological order, i.e. the reservations are ordered with respect to the starting time of the reservations.

How is this pattern implemented?

The strategy pattern is implemented in our system in the following manner:
The strategy interface or in the case of this implementation, the ReservationSortingStrategy interface defines all the operations. In this implementation, an example of an operation is returning the next Reservation. This operation can be carried out using different behaviors implemented through concrete classes that implement this common interface.
Furthermore, in our system there is an aggregation link between a BookingSystem, holding all reservations in a list, and the ReservationSortingStrategy interface. Clearly, our BookingSystem needs to contain a sorting strategy implementation to return reservations in any desired order.

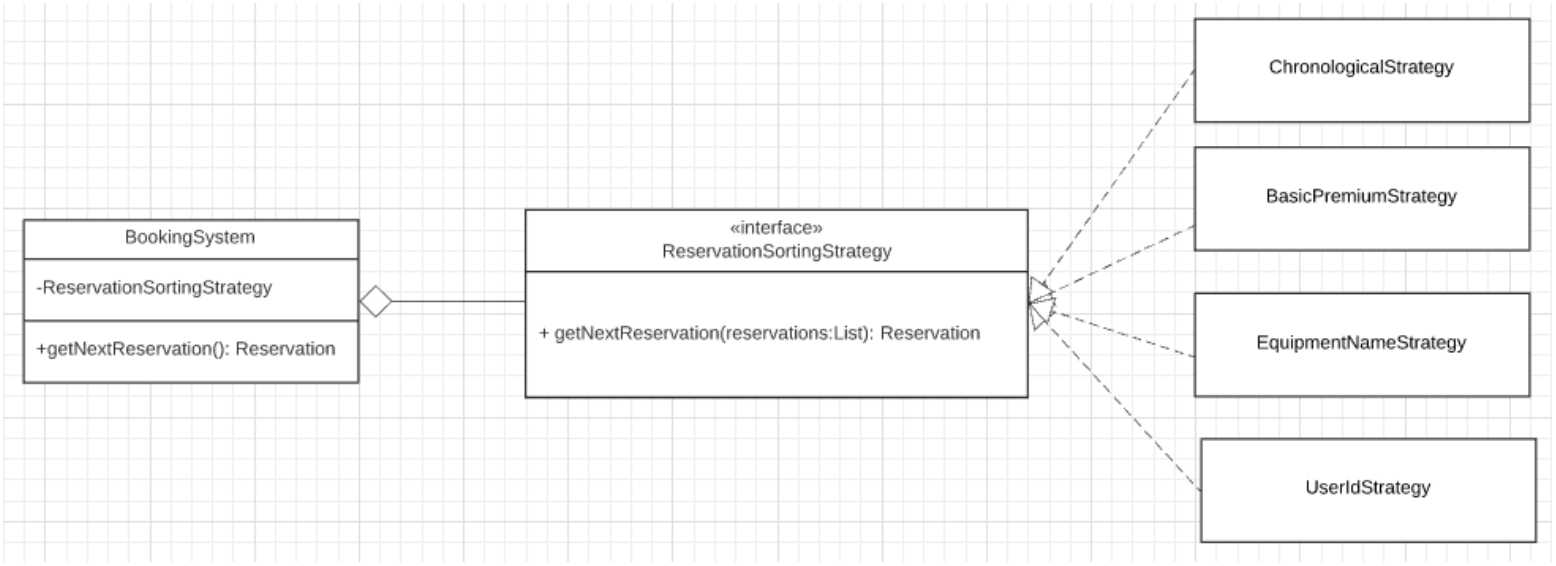
The following is the list of behaviors/strategies we have created:

- Chronological Strategy: Reservations are sorted according to the starting time of the reservations and are returned in descending order, i.e. starting with the most recent reservations.
  If there is maintenance being carried out in the sports centre after a specified time, and some reservations would be affected, these users can be notified beforehand. This strategy is implemented through the ChronologicalStrategy class.

- BasicPremium Strategy: This strategy helps in sorting according to the subscription of the customer; reservations made by the customer having premium subscriptions are returned before the reservations made by users having a basic subscription.
  Since premium users are paying more for services, it gives the admin the capability to serve the premium users before the basic users. This strategy is implemented through the BasicPremiumUserStrategy class.

- EquipmentName Strategy: Reservations with type Equipment are returned first. Then these reservations are sorted in ascending manner based on the name of the equipment. If two reservations have the equipment with the same name, then the reservation with the latest starting time is returned first.
  Using this strategy, the admin can easily infer from the sorted reservations about which equipment will be available at the earliest again. This strategy is implemented through the EquipmentNameStrategy class.

- UserId Strategy: In this strategy, reservations are sorted according to the user IDs in ascending order. This may serve various different purposes for admins, e.g. administrative affairs relating to a certain user. The admin can now get all reservations for the queried user (id) in one go.
  This strategy is implemented through the UserIdStrategy class.



UML class diagram: Strategy Pattern

2. Chain of responsibility pattern

Why is this pattern implemented?

When a reservation is created for equipment or a sports room, several different conditions need to be satisfied to confirm this reservation.
Each condition is checked separately by passing the created reservation through a chain of validators, each responsible for checking for a certain condition.
This reduces coupling between a sender and a request to its receiver, since we issue the initial request to one of multiple validators, without specifying the receiver explicitly. The order of the chain may interchange.

Similar to the Strategy pattern, this pattern adheres to the “Open-Close Principle” as well. We can extend our system with many more concrete validators, without having to change production code.

As long as all checks pass, the next handler will be called by the current validator, to check the respective condition. On the other hand, if any of the conditions in the chain is violated, an appropriate exception will be raised, and the remainder of the chain will be aborted: the reservation is declared unsuccessful.
The respective exception will inform the client about what condition has been violated, and in which step of the chain.

How is this pattern implemented?

In our system, the chain of responsibility consists of one base validator, which contains logic that is shared amongst our three concrete validators: UserReservationBalanceValidator, SportFacilityAvailabilityValidator and TeamRoomCapacityValidator.
All these validators have a common interface (API): ReservationValidator.
Moreover, all concrete handlers allow for dependency injection in their respective constructors, to guarantee testability using mocks and stubs.

The first handler in our chain is UserReservationBalanceValidator, which checks whether or not customers have exceeded their daily reservation limit for sports rooms. Thus, this validator will count all reservations for sports rooms on the selected day (from 00:00 to 23:59), and check whether or not the customer’s limit has been exceeded.
Customers having a basic subscription cannot make more than 1 reservation per day, whereas customers having a premium subscription can make up to 3 reservations per day. For group reservations, this will be checked for each individual member, since we store separate reservations per member.

If the first condition passes, this handler will call our next validator: SportFacilityAvailabilityValidator.
This handler will check whether the reserved sports room or equipment is available for reservation i.e. the sports room and equipment are available only between 16:00 and 23:00. A user can not reserve them outside these designated hours.

Furthermore, this validator is responsible for checking whether or not the reserved sports room or equipment is not in use during the selected time slot (of 1 hour).

In the case of a *sports room* reservation, this is done by querying the reservation database for existing reservations for the same sports room (id) and same starting time. If this is the case, the condition is violated.
On the other hand, if the latter check does pass, we still have to communicate with our Sports Facility microservice to check whether the passed id for the sports room exists. The reason why we do this check afterwards, is because we want to keep the amount of communication between microservices over the network low.
We can namely discard this second request to the Sports Facilities microservice if the previous check was violated.

In the case of *equipment* reservations, we assume the user selects some equipment name, whereafter our system looks for the first existing and available instance of that equipment name. (Note that we do not store equipment names as strings in reservations, only sport facility ids, either for a sports room / equipment / lesson)
This is done by communicating with the Sports Facility microservice, *before* creating the reservation object to be passed through the chain. Now the reservation object to be passed through the chain will either store a valid equipment id, or -1 if there was no existing or available instance of the specified equipment name found. Then, the validator will check this stored id to verify whether available equipment has been found.

Our last validator in the chain, the TeamRoomCapacityValidator, merely applies to reservations for sport rooms, not equipment.
It checks the compatibility of the reserved *sports room (hall/field*) capacity with the group size of the customers who want to reserve that *sports room.*
In the case of a single customer, the “group size” is by default equal to one.

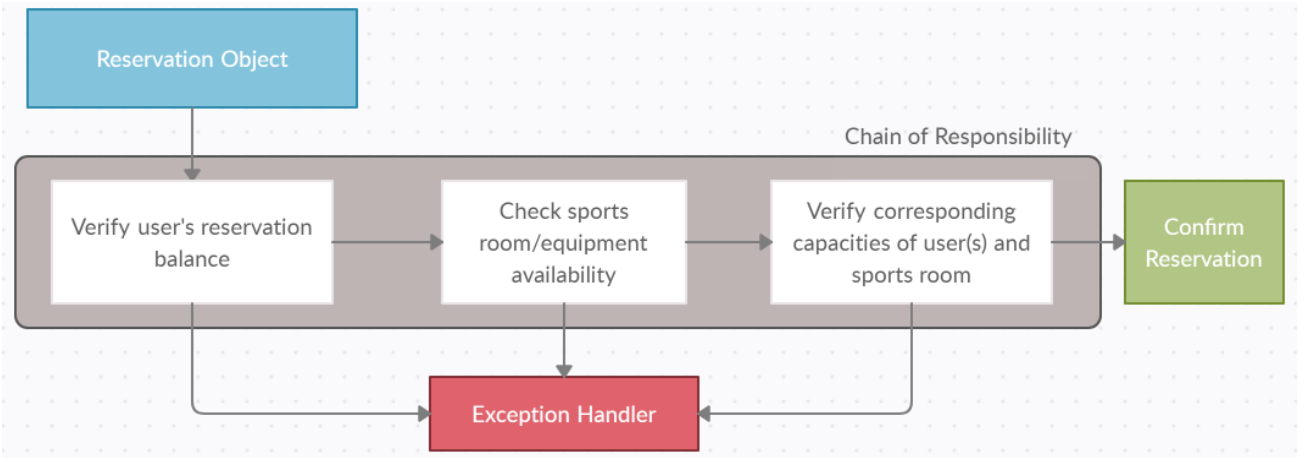
Whereas *sport halls* can entertain multiple sports, a *sports field* is tied to one certain sport. Thus, in the case of a *sport field* being reserved, this validator also checks whether the team size adheres to the minimal and maximal team size for the related sport. For soccer, the range may be 5-15 for instance.

Specifically, this validator checks sub-conditions:

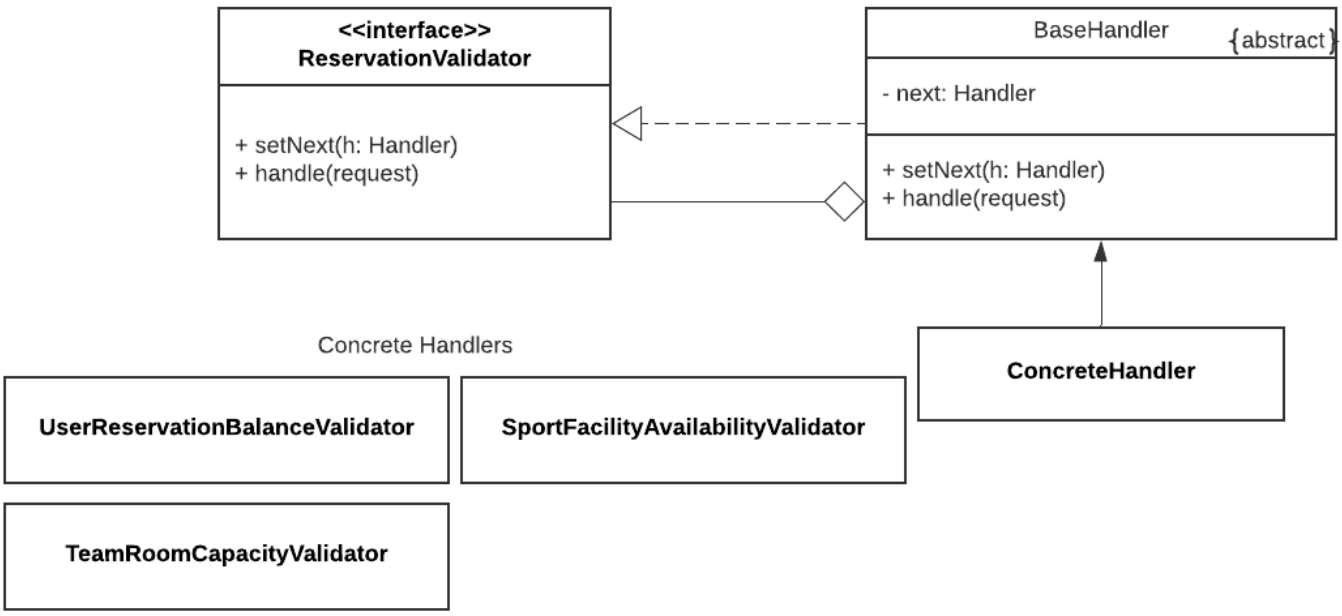
1. For a *sports field,* which is tied to a team sport, the customer(s) capacity has to be at least the minimum team size and atmost the maximum team size.

2. The group size has to be at least the minimum capacity of the *sports room (field/hall)* and atmost the maximum capacity of the *sports room.*

If these two sub-conditions are satisfied, then the reservation is made successfully.



Generic diagram: Chain of Responsibility Pattern



UML class diagram: Chain of Responsibility Pattern