ASSIGNMENT 2

Group 31b

Alexandra-Ioana Neagu, Bozhidar Andonov, Ferhan Yildiz, Jannick Weitzel, Luuk van de Laar, Tudor-George Popica

1. Code metrics

For this assignment we decided to use the tool **CodeMR** to get the code metrics for our project. Our goal would be to improve these metrics based on the information that the tool gave us. When we first ran the tool, we looked through all of the criteria and found that all of them were relatively good, some even had perfect results. We still found some criteria that could be worked upon, though, like the "Lack of Cohesion", "Size" and "Coupling" criterias. A more extensive report of the metrics of the project before and after refactoring can be found at `codemr/project-before` and `codemr/project-after`.

There are 3 metrics that the *Lack of Cohesion* criteria depends on: *Lack of Cohesion of Methods* Metric (**LCOM**, for short), Lack of Cohesion among methods (**LCAM**, for short) and *Lack of Tight Class Cohesion* (**LTCC**, for short). The **LCOM** metric is calculated based on how many **class fields** the methods in that class share, while the **LCAM** metric is calculated based on how many **parameter** types methods share. The **LTCC** metric only measures lack of cohesion in **public** methods. These three metrics together form the *Lack of Cohesion* criteria. It should be as low as possible because that would imply that classes are cohesive (they have only 1 purpose). Therefore, our goal would be to reduce these metrics.

The *Size* criteria metric that **CodeMR** depends on is the *Class Lines of Code* Metric (**CLOC,** for short). The CLOC of a class relates to the number of functional lines of code in a class, meaning lines of the body in a class, not including commented or non empty lines. This metric also indirectly influences the *Complexity* metric, since if a class consists of many lines of code it often becomes incomprehensible and hard to maintain. There's also a *Class-Methods Lines of Code* (**CM-LOC** for short) metric which tracks the same thing, but instead of keeping track of it per class, it checks it per method inside of a class. We will also take a look at the *Number of Fields (**NOF** for short)* and the *Number of methods* (**NOM** for short) metrics since those can also be indicative of a problem with the size of a class. We would like to improve this metric as lengthy methods and classes are usually less maintainable. Big size also usually implies a lack of cohesion. Our goal, therefore, would be to reduce the size metric as much as possible.

The Coupling criteria depends on several other metrics, the relevant ones for our project being: The *Number of Children* metric (**NOC**, for short), The *Coupling Between Object Classes* metric (**CBO**, for short), and the *Access to Foreign Data* metric (**ATFD**, for short). The **NOC** metric is calculated using the size number of

direct subclasses of a class, this indicates how our code reuses itself. The **NOC** metric for our project looked perfectly fine across the board, so we did not have to improve the metric for this assignment. The **CBO** metric is calculated by counting other classes whose methods or attributes are used by a class. Lastly the **ATFD** metric is calculated by looking at the number of attributes of **other** classes that are accessible through a class, directly as well as indirectly. By reducing coupling between classes, we ensure that they do not depend on each other as much.

For methods, we will mostly be trying to improve their size and readability, without too much dependency on **CodeMR** as it does not report any method-level metrics (apart from *Coupling between object classes*). That is why we have decided to split and extract methods which are longer than 30 lines or which have a high cyclomatic complexity (above 5), as both of these are known to reduce readability severely. We will also be trying to improve methods which have scores of low-medium or higher in any of the 4 general criteria **CodeMR** presents.

For classes, we will be taking a more thorough look at what metrics **CodeMR** provides us with. We do not have concrete numbers to base ourselves on (as we feel it strongly depends on the class that needs fixing) but we will only be working on classes, whose metric categories have a severity of low-medium or higher to ensure that something has indeed improved.

2. Refactoring

To improve the metrics that we have explained in the first part of this document, we have refactored 5 classes and 5 methods. In this part we will be diving into the types of refactoring operations that we ended up applying on the identified methods and classes.

2.1 Method refactorings

The following five methods have been refactored mainly because of their size and functionality which could be extracted out of the method and in the class itself.

2.1.1 The `prehandle` method in `AuthenticationInterceptor` in the `reservations` microservice

This method is used for checking whether requests are authenticated before they are forwarded to the reservations microservice. It consists of two parts - the first one is extracting the security token that the user has provided and performing checks on its validity. The second one is making a decision based on the role of the user, provided that the token is valid. Therefore the method could easily be split into two parts, by **extracting** some of the logic to another method in the same class. This has reduced the **LOC** metric for the method.

2.1.2 The `canTeamBook` method in `TeamService` in the `users` microservice

This method is used to check if all of the members of a team aren't exceeding their daily maximum of reservations. It is mainly used when a team wants to book a reservation. We found that there were relatively a lot of lines of code in the method, so we ended up taking a part of the method, the part that checks if two dates are the same date, and we **extracted** it to a different method called `checkIfSameDate`. We've made this change to make the method more comprehensible (Extract Method Refactoring). This reduced the **CM-LOC** metric.

2.1.3 The `getAllAvailableEquipment` method in `EquipmentService` in the `reservations` microservice

This method is responsible for retrieving available equipment at a given time, provided a string of the time. It consists of two parts - first, the string should be converted to a Timestamp object. After that is done, the database should be checked for the equipment that is booked during that time and remove it from the available equipment. So we can split the functionality in two parts again. That is why, two new methods have been introduced - `parseStringToTimestamp` and `getRemainingEquipmentForTime`. This has reduced both the size and the coupling of the method. Here is a screenshot of the metrics before and after the refactoring.

Name	Complexity	Coupling	Size	Lack of Cohesion	CBO
project					
reservations.services	low	low	low	low	
EquipmentService	low	medium-high	low-medium	low-medium	14
EquipmentService(EquipmentRepository, R	low	low	low	low	3
addEquipment(String, int, String): void	low	low	low	low	2
bookEquipment(UUID, UUID, String, int): v	low	medium-high	low	low	10
deleteEquipment(UUID): void	low	low-medium	low	low	5
getAllAvailableEquipment(String): List	low	medium-high	low	low	9
getAllEquipment( ): List	low	low	low	low	1
getEquipment(UUID): Equipment	low	low	low	low	2
removeEquipmentBooking(UUID, UUID): v	low	low	low	low	2
updateEquipment(UUID, String, int, String):	low	low-medium	low	low	4

Before

Name	Complexity	Coupling	Size	Lack of Cohesion	CBO
project					
reservations.services	low	low	low	low	
EquipmentService	low	medium-high	low-medium	low-medium	14
EquipmentService(EquipmentRepository,	low	low	low	low	3
addEquipment(String, int, String): void	low	low	low	low	2
bookEquipment(UUID, UUID, String, int):	low	medium-high	low	low	10
deleteEquipment(UUID): void	low	low-medium	low	low	5
getAllAvailableEquipment(String): List	low	low	low	low	0
getAllEquipment( ): List	low	low	low	low	1
getEquipment(UUID): Equipment	low	low	low	low	2
getRemainingEquipmentForTime(Timest	low	low-medium	low	low	4
parseStringToTimestamp(String): Timest	low	low-medium	low	low	5
removeEquipmentBooking(UUID, UUID):	low	low	low	low	2
updateEquipment(UUID, String, int, Strin	low	low-medium	low	low	4

After

2.1.4 The `changePasswordUser` method in `MyUserDetailService` in the `authentication` microservice

The method `changePasswordUser` had 2 recurring blocks of code, which was the cause of a lot of unnecessary repeating lines. This block has now been made into its own method (`authenticateCredentials`) and this method is called twice where the duplicate lines of code used to be. This prevents having duplicate code, improves readability and decreases lines. This reduced the **CM-LOC** and coupling metrics.

Name	Complexity	Coupling	Size	Lack of Cohesion	CBO
project					
authentication.services	low	low	low	low	
MyUserDetailService	low-medium	medium-high	low-medium	medium-high	16
activation(ActivationRequest): void	low	low-medium	low	low	4
addAdmin(String, AdminRequest): void	low	low-medium	low	low	5
addUserCredential(UserCredential): UserCredential	low	low	low	low	2
authenticate(AuthenticationRequest): String	low	low-medium	low	low	6
authenticateCredentials(String, String): boolean	low	low	low	low	3
changePasswordUser(String, PasswordRequest): void	low	low-medium	low	low	5
deleteUser(String, DeletionRequest): void	low	medium-high	low	low	7
deleteUserCredential(UserCredential): void	low	low	low	low	2
getPermissions(String, JwtUtil): String	low	low	low	low	2
loadUserByUsername(String): UserCredential	low	low	low	low	3
registerUser(RegistrationRequest): void	low	low-medium	low	low	5

Before

Name	Complexity	Coupling	Size	Lack of Cohesion	CBO
project					
authentication.services	low	low	low	low	
MyUserDetailService	low-medium	medium-high	low-medium	medium-high	16
activation(ActivationRequest): void	low	low-medium	low	low	4
addAdmin(String, AdminRequest): void	low	low-medium	low	low	5
addUserCredential(UserCredential): UserCredential	low	low	low	low	2
authenticate(AuthenticationRequest): String	low	low-medium	low	low	6
changePasswordUser(String, PasswordRequest): void	low	medium-high	low	low	7
deleteUser(String, DeletionRequest): void	low	medium-high	low	low	7
deleteUserCredential(UserCredential): void	low	low	low	low	2
getPermissions(String, JwtUtil): String	low	low	low	low	2
loadUserByUsername(String): UserCredential	low	low	low	low	3
registerUser(RegistrationRequest): void	low	low-medium	low	low	5

After

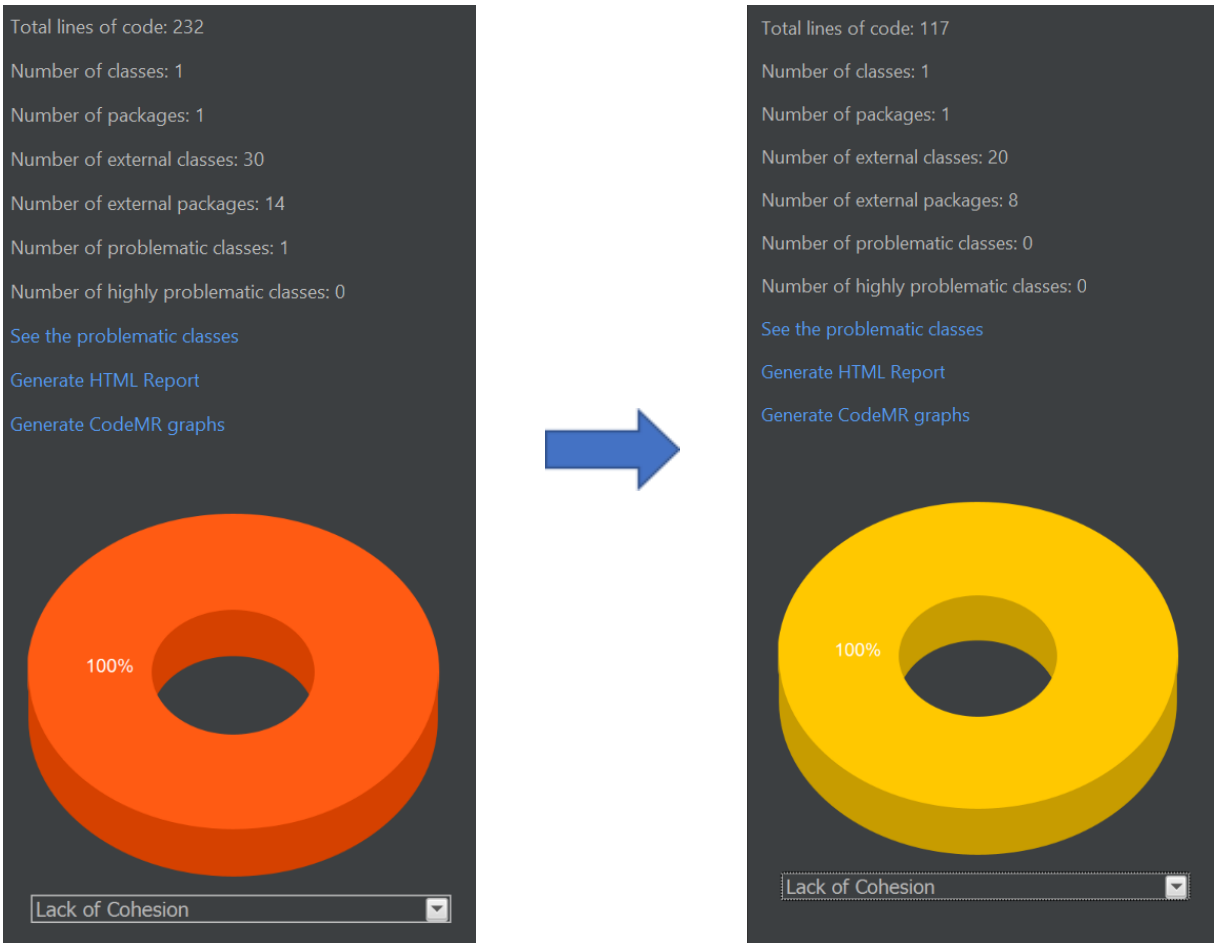
2.1.5 The `doFilterInternal` method in `JwtRequestFilter` in the `authentication` microservice

We have removed some logic from the `doFilterInternal` method that was doing what Spring security usually does. We have **extracted** it to a different method of the same class and the original method now calls the newly created one. This reduces the size and complexity of the method(s).

2.2 Class refactorings

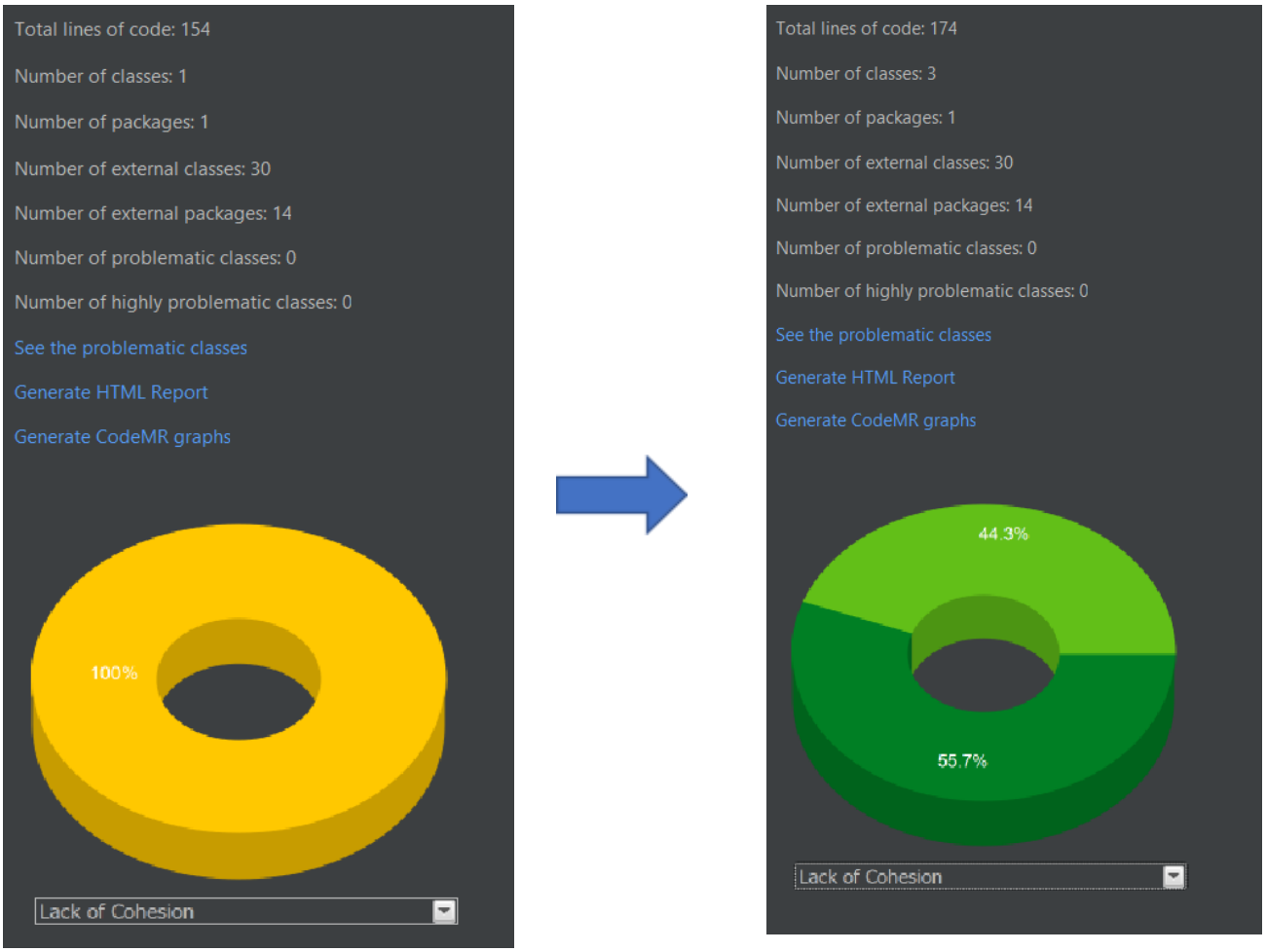
2.2.1 The `AuthenticationAPI` class in the `authentication` microservice

At first, we used to have the implementation (or business logic) of all the authentication endpoints in its API class. Later we realized this was not ideal, as now the API class had multiple responsibilities, which had reduced its cohesion. That is why we decided to refactor the class. We ended up moving the business logic that was in the API to the relevant services. This improved the cohesion metrics by a lot, as seen from the graphs below.



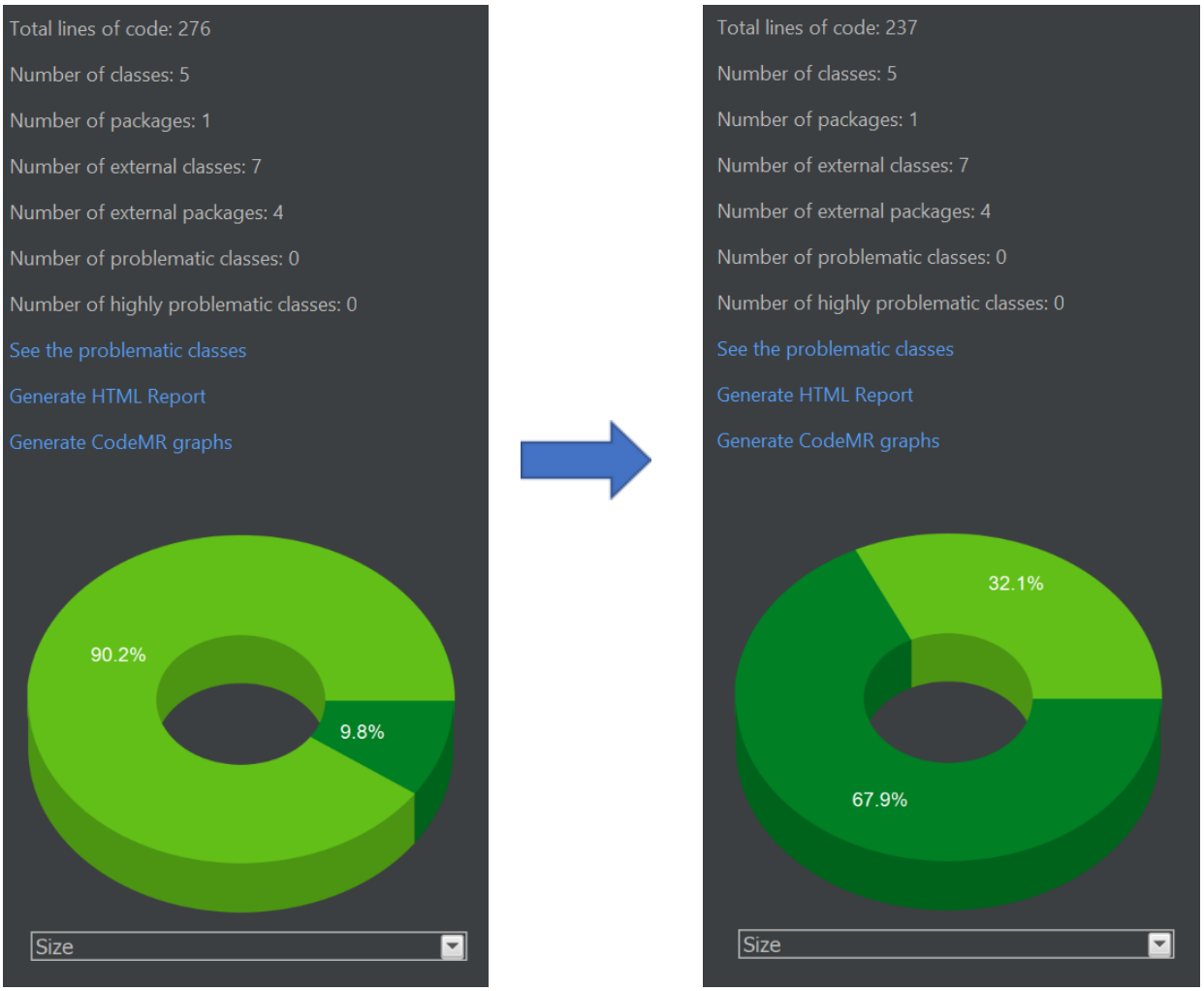
2.2.2 The `myUserDetailService` class in the `authentication` microservice

We improved the cohesion of myUserDetailsService by splitting it into 3 different classes, as its responsibilities were in different ‘fields’, so to speak. This sadly does not improve the coupling metric, as API classes now require multiple services to function correctly. Cohesion has been greatly improved among the newly created classes as can be seen from the graph below.



2.2.3 The `Bookable` class in the `reservations` microservice

For the Bookable class, we removed the field “description”, as it was redundant and we never actually used it. It introduced a lot of bloat code in the constructors of the Bookable classes. Not only that but this propagated to its child classes, meaning there were a lot of unnecessary lines of code. Therefore, we have removed all constructors that once featured that field. We have also removed the relevant getters and setters. We adjusted the tests accordingly, as well. Now the class(es) is more maintainable and testable. Therefore, the size metrics have been improved - the **LOC** and **CLOC** metrics in particular (but **NOF** and **NOM**, as well).



2.2.4 The `LessonService` class in the `reservations` microservice

We found that we had a lot of duplicate code instances for parsing strings to timestamps which introduced quite a lot of duplicate code. That is why we have **extracted** all of that functionality to a method in the same class - namely, to the method `parseStringToTimestamp` . We have then removed duplicate code from the three methods that actually converted strings to timestamps and have replaced it with a call to this method. You can see that different metrics for the different methods (therefore, for the class overall) have been improved below.

Name	Complexity	Coupling	Size	Lack of Cohesion	CBO
project					
reservations.services	low	low	low	low	
LessonService	low-medium	medium-high	low-medium	low-medium	18
LessonService(	low	low-medium	low	low	4
addLesson(Stri	low	medium-high	low-medium	low	9
bookLesson(U	low	very-high	low	low	11
deleteLesson(L	low	medium-high	low	low	7
getAllAvailable	low	medium-high	low	low	8
getAllLessons(	low	low	low	low	1
getLesson(UUI	low	low	low	low	2
getSportsFacilit	low	low	low	low	2
removeLessonF	low	low	low	low	3
updateLesson(	low	medium-high	medium-high	low	9

Before

Name	Complexity	Coupling	Size	Lack of Cohesion	CBO
project					
reservations.services	low	low	low	low	
LessonService	low-medium	medium-high	low-medium	low-medium	18
LessonService(	low	low-medium	low	low	4
addLesson(Stri	low	low-medium	low	low	6
bookLesson(U	low	very-high	low	low	11
deleteLesson(L	low	medium-high	low	low	7
getAllAvailable	low	low	low	low	3
getAllLessons(	low	low	low	low	1
getLesson(UUI	low	low	low	low	2
getSportsFacilit	low	low	low	low	2
parseStringToT	low	low-medium	low	low	4
removeLessonF	low	low	low	low	3
updateLesson(	low	low-medium	low-medium	low	6

After

2.2.5 Some of the validator classes in the `reservations` microservice

We did not introduce this refactoring in this assignment but it is something we did during the project itself which we believe counts as a nice change. We had two different validators for whether a time is valid - `EquipmentTimestampValidator` and `LessonTimestampValidator`. They had really similar business logic, however and were checking multiple things - whether the time was in the future, whether it could be parsed and whether it was in a predetermined range. This introduced a lot of code duplication and lack of cohesion in these classes. So that is why we extracted the functionality in three different validators - `FutureTimestampValidator`, `StringToTimestampValidator` and `TimestampWithinBoundsValidator`.