Design Patterns

2.1

Factory Design Pattern

We implemented this design pattern in the Voting microservice. The microservice handles the Voting class, which is split into 2 subclasses - RuleVoting and Election. When creating a Voting, we have the option to create an Election, a RuleVoting about a new rule, or a RuleVoting about an amendment. For this purpose, we created the VotingFactory class, which handles the logic of when to create which Voting, based on parameters, and returns the new object. We use this because the two subclasses are very similar in scope and functionality. Each Voting has a list of votes with userIds and aim (in our case a string for “for/against/abstain” for the RuleVoting and a candidateId for the Election). There are plenty of other similar or shared parameters but the main difference lies in logic and how we deal with the data, especially with how it affects the association. Moreover, it allows for more Voting types to be easily added in the future.

Facade Design Pattern

We implemented this design pattern in the Voting microservice. The Voting microservice has a lot of logic inside of it. It has all the methods with the core logic of the voting functionalities, VotingFactory, which creates the different Voting object, Election and RuleVoting classes, their respective Repositories, a Scheduler, various Converters and Exceptions, the list goes on… Providing all of these functionalities to the VotingController separately would be very inconvenient and difficult to deal with, while also cluttering the Controller class. Therefore, we decided to make the VotingService. It encapsulates all of these functionalities and creates a facade of simple methods for the VotingController to use.

The VotingService implements the functionalities of the Voting microservice in several methods, where it uses the methods from Election and Rulevoting. It uses the ElectionRepository and the RuleVoting repository to retrieve and store

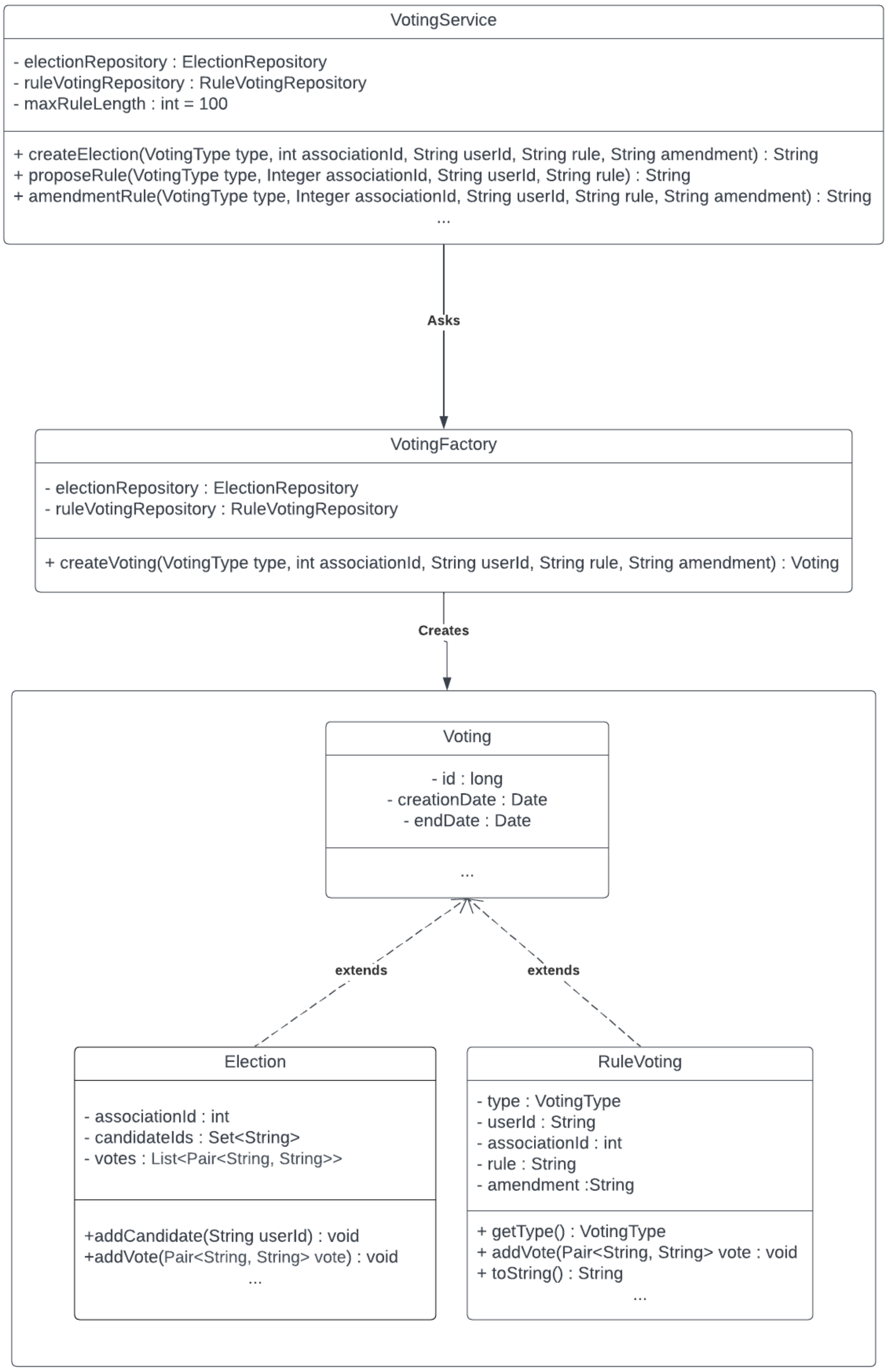
Election/RuleVoting objects. The Schedulers handle the results of Elections/RuleVotings. They run once a fixed amount of time and find the oldest finished Election/RuleVoting still in the respective repository. Then, they make a call to the Association microservice and send all the relevant information. The Association microservice updates the Council/Rule, as well as the History object for the relevant association. Once everything is done, the Scheduler deletes the Election/RuleVoting that was sent from its respective repository. The Converters are used to convert collections stored in the database to a usable format.

With the help of VotingService, the implementation of VotingController can be focused around ResponseEntity, ResponseStatusException and RequestBody, while using the facade of VotingService to provide the necessary logic.

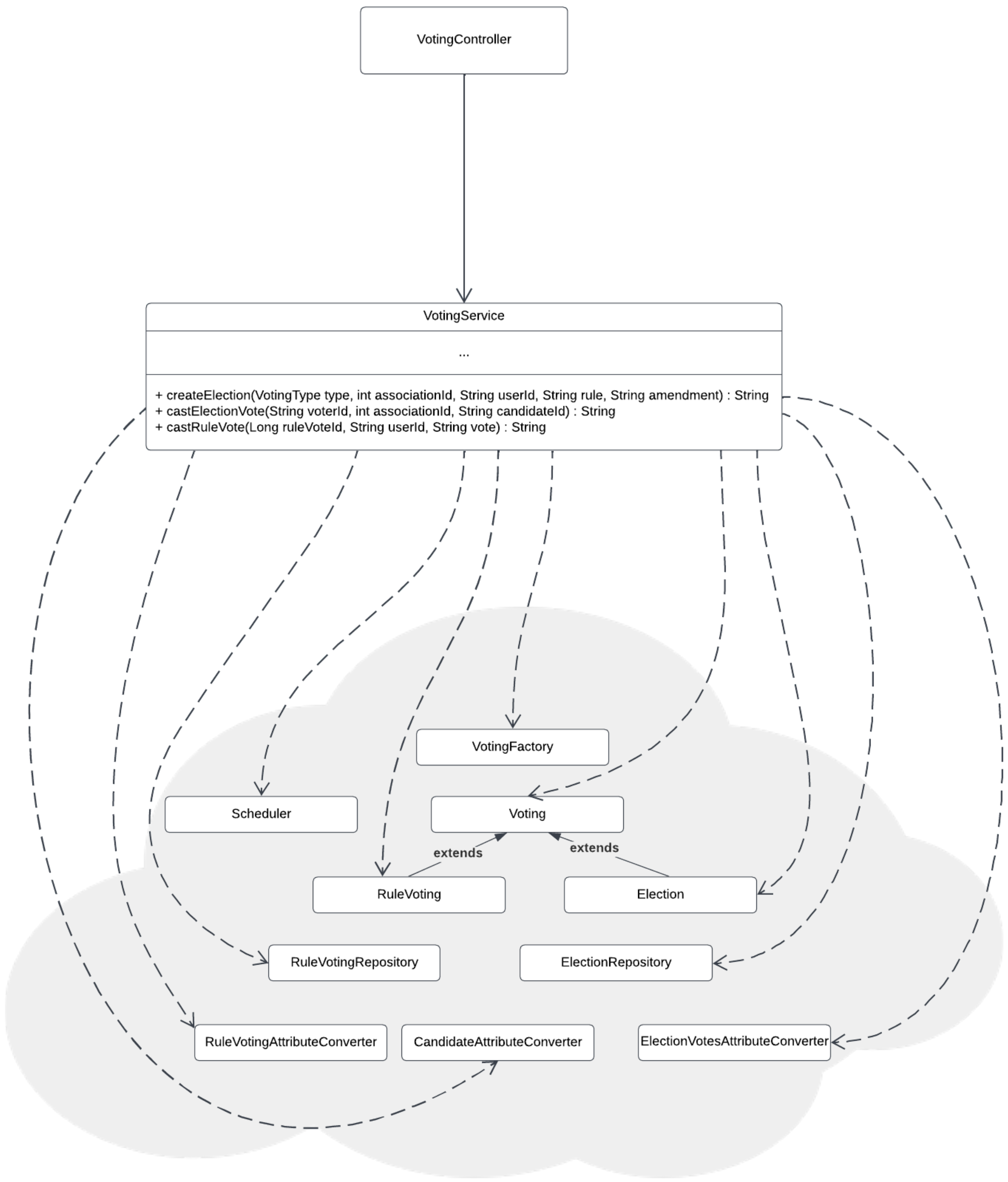
2.2
Class Diagrams

Below, you can see the class diagrams for our design patterns:

Factory


VotingService
- electionRepository : ElectionRepository
- ruleVotingRepository : RuleVotingRepository
- maxRuleLength : int = 100
+ createElection(VotingType type, int associationId, String userId, String rule, String amendment) : String
+ proposeRule(VotingType type, Integer associationId, String userId, String rule) : String
+ amendmentRule(VotingType type, Integer associationId, String userId, String rule, String amendment) : String
...
Asks
VotingFactory
- electionRepository : ElectionRepository
- ruleVotingRepository : RuleVotingRepository
+ createVoting(VotingType type, int associationId, String userId, String rule, String amendment) : Voting
Creates
Voting
- id : long
- creationDate : Date
- endDate : Date
...
extends
extends
Election
- associationId : int
- candidateIds : Set<String>
- votes : List<Pair<String, String>>
+addCandidate(String userId) : void
+addVote(Pair<String, String> vote) : void
...
RuleVoting
- type : VotingType
- userId : String
- associationId : int
- rule : String
- amendment :String
+ getType() : VotingType
+ addVote(Pair<String, String> vote : void
+ toString() : String
...

Facade


VotingController
VotingService
...
+ createElection(VotingType type, int associationId, String userId, String rule, String amendment) : String
+ castElectionVote(String voterId, int associationId, String candidateId) : String
+ castRuleVote(Long ruleVoteId, String userId, String vote) : String
VotingFactory
Scheduler
Voting
extends
extends
RuleVoting
Election
RuleVotingRepository
ElectionRepository
RuleVotingAttributeConverter
CandidateAttributeConverter
ElectionVotesAttributeConverter

2.3

Implementation

The design patterns are implemented in the following paths:
Factory:
voting-microservice/src/main/java/nl/tudelft/sem/template/voting/domain/VotingFactory.java
Facade:
voting-microservice/src/main/java/nl/tudelft/sem/template/voting/domain/VotingService.java