Design Patterns

We decided that the facade design pattern and the chain of responsibility would be most appropriate and improve the design of our architecture the most. We implement the facade design pattern in our users-microservice to route the client requests and encapsulates the applications internal architecture and endpoints and we implement the chain of command design pattern to handle the correct checks and validation at each step for the removal of nodes from the cluster.

Facade

WHY A FACADE?

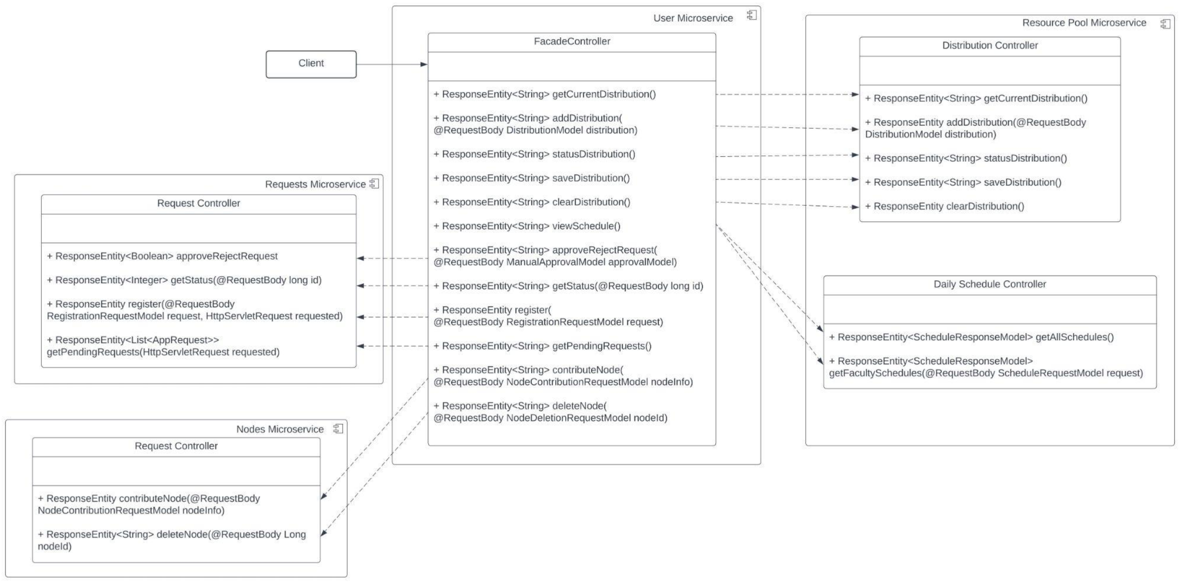
A user can make many requests to perform a diverse set of operations such as contributing a node, creating a faculty, making a request for computing resources, or manually approving requests, etc. This requires many complex processes like authenticating and authorizing the user, and routing the request to the appropriate endpoint at the appropriate microservice. We chose to implement the facade design pattern to create a single entry point for the client, in order to encapsulate the application's internal architecture and hide away the complexity of those services from the external client.

In the facade design pattern, a facade is an object that provides a simplified interface to a larger body of code, such as a class library. A facade is often used to provide a convenient interface for clients to access the functionality of a complex system, without needing to know the details of how that functionality is implemented. Our request gateway serves a similar purpose by providing a single, unified interface for clients to access a set of services while hiding the complexity of those services from the client.

If in the future we would want to improve the scalability and performance of our application this request gateway could cache responses from internal services, reducing the load on those services and improving the overall performance of the system.

The one disadvantage of implementing this design pattern that is of note is that the high availability of this component creates the risk that the facade becomes a development bottleneck. Developers must consistently update the facade in order to expose their services’ endpoints. For a relatively small-scale application like this, the risk of this becoming a significant problem is low and did not present in a significant way during our sprints.

CLASS DIAGRAM


Client
User Microservice
FacadeController
+ ResponseEntity<String> getCurrentDistribution()
+ ResponseEntity<String> addDistribution(@RequestBody DistributionModel distribution)
+ ResponseEntity<String> statusDistribution()
+ ResponseEntity<String> saveDistribution()
+ ResponseEntity<String> clearDistribution()
+ ResponseEntity<String> viewSchedule()
+ ResponseEntity<String> approveRejectRequest(@RequestBody ManualApprovalModel approvalModel)
+ ResponseEntity<String> getStatus(@RequestBody long id)
+ ResponseEntity register(@RequestBody RegistrationRequestModel request)
+ ResponseEntity<String> getPendingRequests()
+ ResponseEntity<String> contributeNode(@RequestBody NodeContributionRequestModel nodeInfo)
+ ResponseEntity<String> deleteNode(@RequestBody NodeDeletionRequestModel nodeId)
Resource Pool Microservice
Distribution Controller
+ ResponseEntity<String> getCurrentDistribution()
+ ResponseEntity addDistribution(@RequestBody DistributionModel distribution)
+ ResponseEntity<String> statusDistribution()
+ ResponseEntity<String> saveDistribution()
+ ResponseEntity clearDistribution()
Daily Schedule Controller
+ ResponseEntity<ScheduleResponseModel> getAllSchedules()
+ ResponseEntity<ScheduleResponseModel> getFacultySchedules(@RequestBody ScheduleRequestModel request)
Requests Microservice
Request Controller
+ ResponseEntity<Boolean> approveRejectRequest
+ ResponseEntity<Integer> getStatus(@RequestBody long id)
+ ResponseEntity register(@RequestBody RegistrationRequestModel request, HttpServletRequest requested)
+ ResponseEntity<List<AppRequest>> getPendingRequests(HttpServletRequest requested)
Nodes Microservice
Request Controller
+ ResponseEntity contributeNode(@RequestBody NodeContributionRequestModel nodeInfo)
+ ResponseEntity<String> deleteNode(@RequestBody Long nodeId)

In the class diagram, you can find our single, unified interface in the center of the diagram. The hidden complexity of those services can be represented as the lines toward the peripherals of the diagram. All calls by the user to any other microservice are routed through the facade controller to the appropriate endpoint in the relevant microservice. Note that the services which manage these complex calls also interact with the authentication microservice. This, however, is not relevant to our diagram because no request from users will directly interact with the authentication microservice and the facade pertains to the external interface the client interacts with.

IMPLEMENTATION

The implementation of this design pattern can be found in the facade directory of the user microservice. The FacadeController manages all the requests by the user and the sendRequestService handles all the complex logic needed to actually send the requests. The routed endpoints can be found in the relevant microservices as indicated in the class diagram.

Path: users-microservice/src/main/java/nl/tudelft/sem/template/users/facade

Chain of Responsibility

WHY CHAIN OF RESPONSIBILITY?

From the domain specification, we know the following must hold for a request to remove a contributing node to the cluster

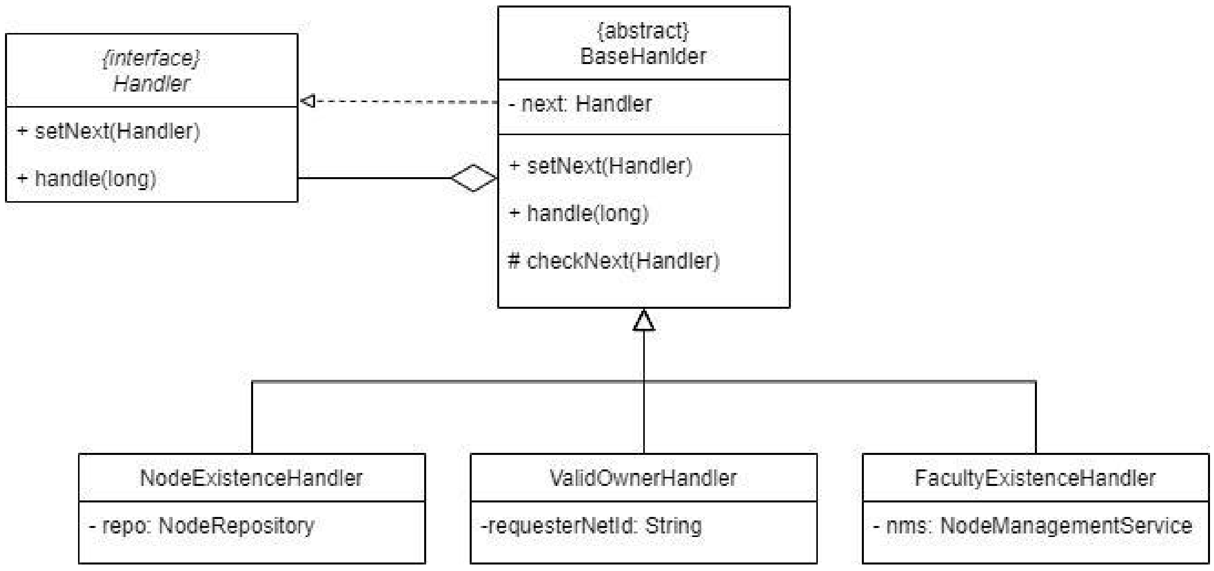
- “The original user who submitted the node can take the node offline from the cluster”.
- “A node cannot be withdrawn the day of the request but should be removed from the cluster starting the next day.”
- “Both the faculty and the node must exist, in order for the node to be taken down.”

Implementing these features would require a complex node removal verification procedure with calls to multiple different microservices. Therefore, the Chain of Responsibility design pattern can encapsulate each step or check of the node, which can make the code easier to understand and maintain and more importantly decouples the sender from the receiver.

Furthermore, the process of removing a node from a computing cluster may vary depending on the owner of the node or the circumstances under which it is being removed. The Chain of Responsibility pattern allows us to create a chain of objects that can handle different types of requests, so you can easily adapt the validation of the node removal requests without affecting the rest of the system. In essence, what this does is improve the flexibility of our system and achieve loose coupling which is very important in handling a request like this that communicates with many different parts of the application and can cause dependency issues.

This flexibility and loose coupling are particularly relevant for our application. This is because we didn't fully implement all functionality related to dropping and picking up jobs after a node is taken down. We chose to deprioritize this issue because we interpreted this area of the domain specification requirements as a "could have" using the MOSCOW prioritization method. Our use of this design pattern, however, means that when we want to update our system to implement the full validity check for requests scheduled using resources from this node we would not have to change the system but could create a new concrete implementation of the handler.

CLASS DIAGRAM:



In our implementation of the chain of responsibility, we have three concrete implementations of an abstract baseHandler which handles the validity of the request in a decoupled fashion, handling one isolated task and/or interaction with another microservice.

IMPLEMENTATION:

In our implementation of the chain of responsibility, we have three concrete implementations of an abstract baseHandler which handles the validity of the request in a decoupled fashion, handling one isolated task and/or interaction with another microservice.

The code for the chain of responsibility design pattern is in the nodes-microservice in the chain directory inside the domain layer directory.

Once the deleteNode request has been routed to the node-microservice the chain is instantiated and executed which works as follows:

In the first step of our chain, the NodeExistenceHandler we verify whether the node can be found in the node repository. If not, an exception is thrown and this is communicated to the user and the chain is terminated. If the node was found the handler checks whether there is a next handler configured and passes on the nodeId to that handler.

The second step of our chain is the ValidOwnerHandler which directly coincides with point 1 listed in our domain specification surrounding the removal of nodes. Here the handler verifies that the netId of the user making the request is the same as the owner of the node. If so it checks whether there is another handler in the chain and passes on the nodeId. Upon failure, the custom exception is thrown and communicated with the user, terminating the chain.

In the third step of our chain, the FacultyExistenceHandler verifies whether the assigned faculty of the node actually exists and is stored in the resource pool repository in the resource pool microservice. This requires a request to be sent to the resource pool microservice and upon receiving a valid response, the FacultyExistenceHandler checks if there is a next handler in the chain and passes on the id of the node. If the faculty in fact does not exist, this is communicated to the user and the chain ends.

Finally, if the chain succeeds, we send a message to the resourcepool-microservice to remove the node from the cluster and the node is removed from the repository in the node repository. Then, the success of the request is communicated to the user.

In the future, we would like to add the following handlers:

- A handler to ensure that it only happens the next day and not the day of the request
- A handler that handles dropping requests for clusters that no longer have enough resources.

This is precisely why we chose a chain of responsibility because the loose coupling allows us to add this functionality without further making changes in the system other than in the chain.

Path: nodes-microservice/src/main/java/nl/tudelft/sem/template/nodes/domain/node/chain