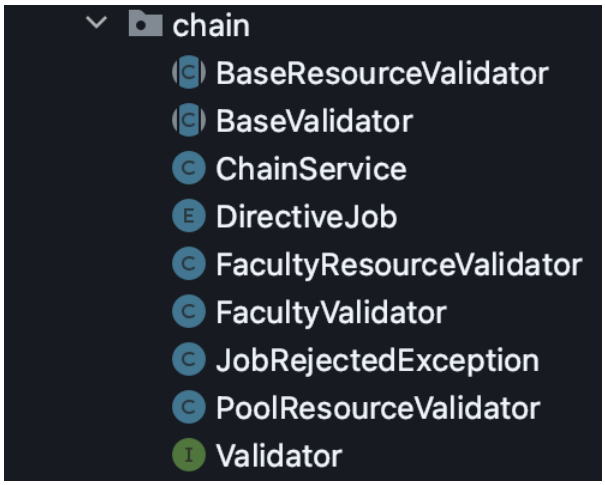
Group-DB-09b-Assignment-1-Part-2

Design Pattern: *Chain of Responsibility*

Natural Language Description:

The reason we chose the Chain of Responsibility design pattern in our system is the approval/rejection of a job before being sent to the scheduler. To be more specific, the design pattern is implemented in Jobs microservice where users are able to create jobs that are initialized in the pending state. Following the client's requirements/scenario, once a Job is created it needs to go through multiple checks in order to be scheduled. Firstly, the faculty account (specific to the job) needs to approve the job. Secondly, we need to check if that specific faculty has enough resources available to handle the job, before or on the preferred date specified by the Job creator. Last but not least, if a faculty does not have enough resources available by itself, we check if all requirements can be satisfied once we add the resources from the free pool. As a result the chain of responsibility contains the three steps described above.

In the project folder, the file structure for chain of responsibility design pattern is as follows:



In the folder "chain" located in Jobs Microservice all the logic of the design pattern is implemented. With the guidelines from the lectures and resources found online, the code is structured in a manner that allows for extensions in an intuitive and simple way, by adding or removing handlers based on new requirements.

Chain of Responsibility is a behavioral design pattern that lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain. The interface “Validator” specifies the methods that need to be implemented by all validators present in the chain.

```
2 usages  1 implementation  mlica
void setNext(Validator handler);

2 usages  3 implementations  mlica
boolean handle(JobChainModel jobChainModel) throws JobRejectedException;
```

More specifically, we have a “setNext” method, which sets the next handler in the chain of responsibility and the “handle” method which does the actual checks at every step. The method “handle” returns true or false, signifying that the Job is eligible for that validator and false otherwise. If the Job is eligible with all validators, then the Job is approved. If a validator fails (the “handle” method returns false) the chain of responsibility stops and the Job is rejected.

Using the “Validator” interface we create an abstract class called “BaseValidator” that is used to derive all the actual validators in the chain of responsibility. The field “next” in “BaseValidator” represents the next handler in the chain.

```
3 usages
private transient Validator next;
```

The “BaseValidator” abstract class also introduces a new method “checkNext” which passes the “responsibility” to the next handler.

```
protected boolean checkNext(JobChainModel jobChainModel) throws JobRejectedException {
    if (next == null) {
        return true;
    }
    return next.handle(jobChainModel);
}
```

For convenience, we added another abstract class called “BaseResourceValidator” which deals with handlers related to resource allocation as presented above.

Finally, we implement the actual validators in “FacultyValidator”, “FacultyResourceValidator” and “PoolResourceValidator”, which are all subclasses of BaseValidator.

```java
public class PoolResourceValidator extends BaseResourceValidator {

    2 usages   mlica *
    @Override
    public boolean handle(JobChainModel jobChainModel) throws JobRejectedException {
        Job job = jobChainModel.getJob();
        List<Faculty> faculty = jobChainModel.getAuthFaculty();
        faculty.add(new Faculty( facultyName: "Pool"));
        LocalDate localDate = LocalDate.now();
        List<FacultyResource> resources = getFacultyResources(faculty, localDate);
        int cpuAvailable = resources.stream().mapToInt(FacultyResource::getCpuUsage).sum();
        int gpuAvailable = resources.stream().mapToInt(FacultyResource::getGpuUsage).sum();
        int memoryAvailable = resources.stream().mapToInt(FacultyResource::getMemoryUsage).sum();
        if (job.getCpuUsage() > cpuAvailable || job.getGpuUsage() > gpuAvailable || job.getMemoryUsage() > memo
            return false;
        }
        return super.checkNext(jobChainModel);
    }
}
```

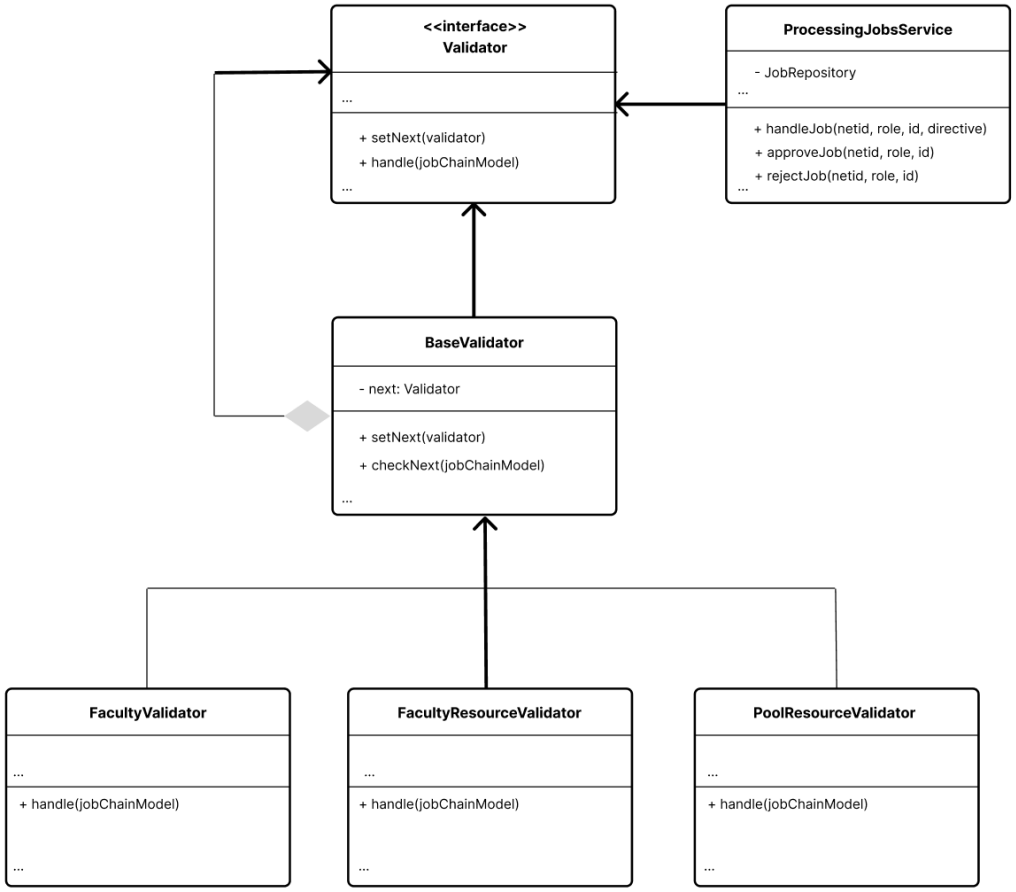
This is the “PoolResourceValidator” which checks if the resources requirements by the job can be satisfied. The “FacultyResourceValidator” checks if the faculty specified by the job has enough resources by itself to handle the requirements. The “FacultyValidator” checks if the user who accessed the endpoint is a faculty account corresponding to the faculty in the Job.

The actual logic for endpoints used to approve/reject a job is in the class “ChainService” which starts the chain of responsibility.

```java
Validator handler = new FacultyValidator();
Validator handler2 = new FacultyResourceValidator();
handler.setNext(handler2);
handler2.setNext(new PoolResourceValidator());
try {
    boolean valid = handler.handle(jobChainModel);
    if (valid) {
        j.setStatus(Status.ACCEPTED);
    } else {
        j.setStatus(Status.REJECTED);
    }
    return j;
} catch (Exception e) {
    throw new Exception(e);
}
```

Here we initialize the validators that will be present in the chain and start by calling the first validator by “handler.handle()”. Depending on the result from the chain of responsibility, the job is accepted or rejected.

Class Diagram:

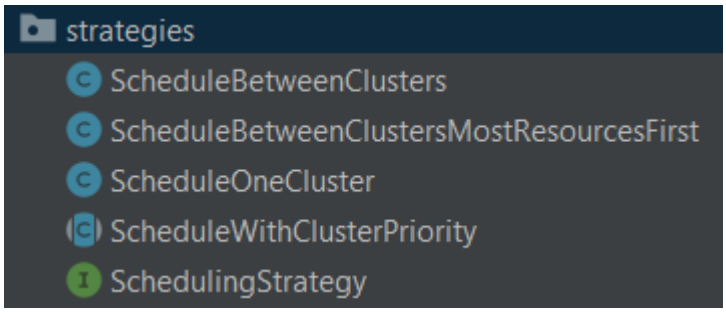


Design Pattern: *Strategy*

Natural Language Description:

The strategy design pattern is a natural choice for the Scheduler microservice. It allows it to use different scheduling algorithms and let the admin choose the appropriate one during runtime.

The application currently has three strategies: *ScheduleOneCluster*, *ScheduleBetweenClusters*, and *ScheduleBetweenClustersMostResourcesFirst*.



The first one tries to schedule a job in only one cluster with resources. The second one includes the possibility to split the job into all available clusters. The last one works like the second one, but it prioritizes clusters with the most free resources - it can be used to fill all clusters as equally as possible. It extends the *ScheduleWithClusterPriority* abstract class that defines the *scheduleBetween* method that accepts a *Comparator* to manipulate the order in which clusters will be considered.

All of the strategies implement the *SchedulingStrategy* interface, defined as shown below:

```
public interface SchedulingStrategy {
    List<ScheduledInstance> scheduleBetween(ScheduleJob job, LocalDate start, LocalDate end);
}
```

It is a small interface with one important function that is used in the *ProcessingJobsService*. This service has a *SchedulingStrategy* field which allows it to reference different strategies with only one field and change them in runtime:

```
@Service
public class ProcessingJobsService {

    private final transient ScheduledInstanceRepository scheduledInstanceRepository;
    private final RestTemplate restTemplate;
    private final ResourceGetter resourceGetter;
    private SchedulingStrategy schedulingStrategy;
```

This object is later used for the *scheduleBetween()* call to obtain a list of objects of *ScheduleInstance* class that represent job's resources scheduled in chosen clusters:

```
List<ScheduledInstance> scheduledInstances =
        schedulingStrategy.scheduleBetween(j, LocalDate.now().plusDays(possibleInXdays), j.getScheduleBefore());
```

Finally, the admin is allowed to change the scheduling strategy through the handy */change-scheduling-strategy* endpoint:

```java
@PostMapping("/change-scheduling-strategy")
public ResponseEntity<String> changeSchedulingStrategy(@RequestBody ChangeSchedulingStrategy
                                                        changeSchedulingStrategy) {
    if (!authManager.getRole().toString().equals("ADMIN")) {
        return ResponseEntity.status(HttpStatus.UNAUTHORIZED).build();
    }

    try {
        processingJobsService.setSchedulingStrategy(changeSchedulingStrategy.getStrategy());
    } catch (Exception e) {
        throw new ResponseStatusException(HttpStatus.BAD_REQUEST, e.getMessage(), e);
    }

    return ResponseEntity.ok("Changed");
}
```

It calls the *setSchedulingStrategy* method in the *processingJobsService* that chooses a strategy based on a given name:

```java
public void setSchedulingStrategy(String strategy) throws InvalidStrategyNameException {
    switch (strategy) {
        case "one-cluster":
            setSchedulingStrategy(new ScheduleOneCluster(resourceGetter, scheduledInstanceRepository));
            break;
        case "multiple-clusters":
            setSchedulingStrategy(new ScheduleBetweenClusters(resourceGetter, scheduledInstanceRepository));
            break;
        case "multiple-clusters-most-resources-first":
            setSchedulingStrategy(new ScheduleBetweenClustersMostResourcesFirst(resourceGetter,
                    scheduledInstanceRepository));
            break;
        default:
            throw new InvalidStrategyNameException(strategy);
    }
}
```

Class Diagram: