**Aspect Oriented Programming (AOP):**

**AOP** : Aspect based programming , application can be logically divided using

cross cutting concerns. e.g. Spring AOP, AspectJ

**Aspect** : class with advice and pointcut definition

**Pointcut** : is an expression which decide where to execute an advice

**Advice** : an action taken before/after/around the execution pointcut

**Join point** : point of application execution where we can apply aspect

**Target object :** on which aspect is applied (In this case :EmployeeService.java)

**Proxy**: It is an object that is created after applying advice to a target

object is called proxy. AOP implements the JDK dynamic proxy

For Example refer below code:

// pointcut (the expression language parameter)

// join point will be "before execution of all methods" in EmployeeService class

@Before(value = "execution(\* com.sujeetpoc.springbootfeatures.service.EmployeeService.\*())")

**public** **void** beforeCallingEmployeeMethods() {

***log***.info("Aspect- Before Creating Employee..");

}

(Refer EmployeeAspect Class in code)

**Exception handling using @ExceptionHandler & @ControllerAdvice :**

The @ControllerAdvice and @ExceptionHandler annotations are powerful tools for handling exceptions in Spring Boot.

The @ControllerAdvice annotation is used to define a class that will be called whenever an exception is thrown in your application. This class can contain multiple methods, each of which is annotated with the @ExceptionHandler annotation and is responsible for handling a specific exception.

@ControllerAdvice  
public class GlobalExceptionHandler {  
  
 @ExceptionHandler(value = MyCustomException.class)  
 public ResponseEntity<ErrorResponse> handleMyCustomException(MyCustomException ex) {  
 ErrorResponse errorResponse = new ErrorResponse(ex.getMessage());  
 return new ResponseEntity<>(errorResponse, HttpStatus.BAD\_REQUEST);  
 }  
  
}

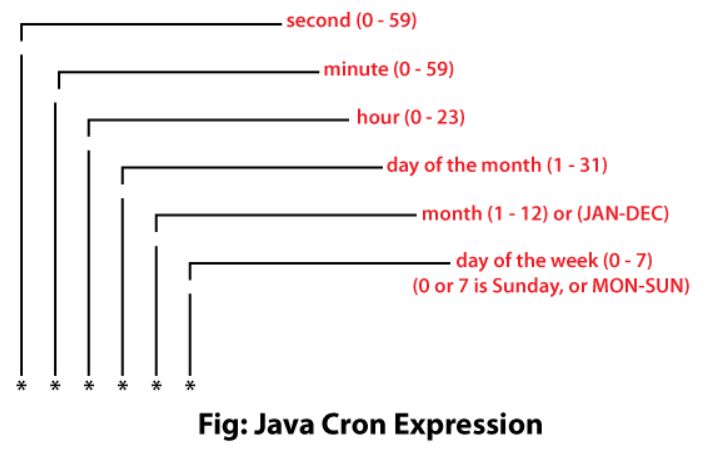
In this example, the GlobalExceptionHandler class is annotated with @ControllerAdvice, which means it will be called whenever an exception is thrown in the application. It contains a single method, handleMyCustomException, which is annotated with @ExceptionHandler and is responsible for handling the MyCustomException exception. The method returns a ResponseEntity object, which is a special object that represents an HTTP response, including the status code and the body of the response.

( Refer “GlobalExceptionHandler” class in code)

**Spring Scheduling + Schedlock:**

Cron is a utility that schedules a task. It allows user to schedule a task periodically at specified time, date, month, weak, and year.

Java CRON expression has only 6 fields as shown below:



You can use following or any online website to generate CRON expression:

[Free Online Cron Expression Generator and Describer - FreeFormatter.com](https://www.freeformatter.com/cron-expression-generator-quartz.html)

Steps:

* Specify @EnableScheduling annotation in the Spring Boot application class, which will facilitates Spring Boot with scheduled task execution capability.
* Create a @Component class Scheduler which defines the method scheduleTask() for scheduling a task using the @Scheduled annotation.

(Refer SampleScheduler Class in code)

When our application runs on a single instance we will have no problem as the execution happens only once. But when our application runs in a distributed environment with multiple instances running in parallel, our scheduled jobs are executed without consistency. It will result in redundant or inconsistent data. We use Shedlock to solve this issue.

Shedlock uses persistent storage that connects to all the instances to store the information about the scheduled task. There are multiple ways to implement this LockProvider as Shedlock has support for numerous databases like MongoDB, PostgreSQL, Redis, MySQL, etc.

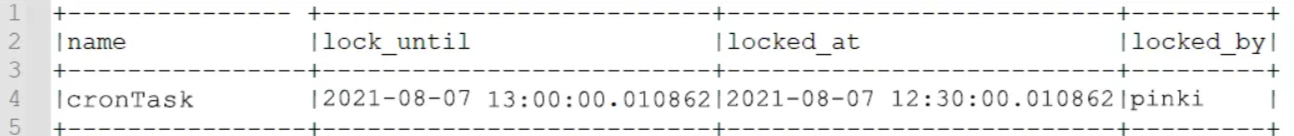
The table that Shedlock uses inside the database to manage the locks is straightforward. It has only had four columns:

name : A unique name provided by the user for scheduled job

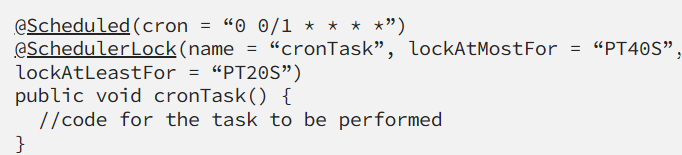
lock\_until : The time till the current execution of the job is locked

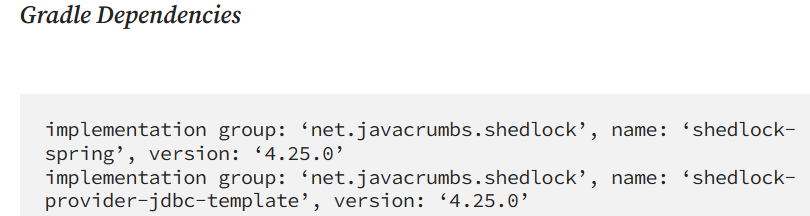
locked\_at : The timestamp when the instance acquired the lock

locked\_by : The identifier who acquires the lock



When we use *lockAtMostFor* as a parameter in the locking we can prevent the task to go beyond the time specified but let’s say we have a scenario where the task gets completed in a short time like 10 sec and we have multiple instances whose clock is not synced then it such scenario the instance will release the lock and the other one can acquire the lock to perform the same redundant task. For such a case we use the *lockAtLeastFor* parameter to prevent the job from running the same task at the same time multiple times. Here is an example of such a scenario:





**Spring Cloud Discovery With Eureka Server and Client:**

Eureka Server is service discovery for your microservices, where all client applications (Eureka Clients) can register by themselves and other microservices look up the Eureka Server to get independent microservices to get the job complete.

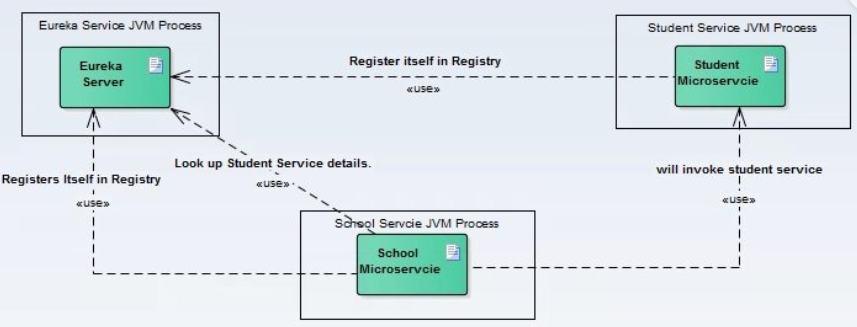
When a service is registered with Eureka Server it keeps sending heartbeats for certain intervals. If the Eureka server didn’t receive a heartbeat from any service instance it will assume the service instance is down and take it out from the pool.

Steps:

Add eureka server dependencies in server app. We need to add @EnableEurekaServer annotation to make our SpringBoot application a Eureka Server-based Service Registry.

Then run this app which will work as eureka service registry. It's dashboard can be accessed at localhost:8761 address.

Then create eureka client. We need to add @EnableDiscoveryClient annotation to make our SpringBoot application a Eureka Server-based Service Registry. Specify eureka server details in properties file and start app, it will register to eureka service registry.

****

(Refer EurekaServer and Springbootfeatures repositories in code)