**Scheduling in SpringBoot**

Spring Boot provides easy-to-use scheduling capabilities through the @Scheduled annotation, which allows you to define methods that should be executed on a regular basis. This annotation is part of the Spring Framework's scheduling support and can be used to schedule tasks within a Spring Boot application without needing to configure a separate task scheduler.

**Here's an overview of Spring Boot scheduling:**

* Enable Scheduling: To enable scheduling support in your Spring Boot application, you typically need to annotate your main application class with @EnableScheduling. This annotation ensures that Spring Boot will scan for @Scheduled annotations and execute the scheduled tasks accordingly.
* Define Scheduled Tasks: You can define scheduled tasks by annotating methods with @Scheduled in any Spring-managed bean. These methods will be invoked based on the specified schedule.

**Schedule Configuration:**

* Fixed Rate: Use the fixedRate attribute to specify the interval between method invocations, in milliseconds.
* Fixed Delay: Use the fixedDelay attribute to specify the delay between the end of the last invocation and the start of the next invocation, in milliseconds.
* Cron Expression: Use the cron attribute to define more complex schedules using cron expressions.

**Fixed Rate vs Fixed Delay**

In Spring's scheduling mechanism, specifically when using the @Scheduled annotation, you have the choice between two attributes to control the timing of task execution: fixedRate and fixedDelay. Each attribute has its own behavior, and understanding the difference between them is crucial for defining the desired scheduling behavior.

**Fixed Rate (fixedRate):**

fixedRate defines the interval between the start times of consecutive invocations of the annotated method, regardless of when the previous invocation completed.

This means that the method will be executed at a fixed rate, irrespective of how long the method takes to execute.

If a task takes longer to execute than the specified interval, subsequent executions will start immediately after the previous one completes, potentially resulting in overlapping executions.

**Fixed Delay (fixedDelay):**

fixedDelay defines the delay between the completion of one execution and the start of the next execution.

With fixedDelay, each execution of the annotated method will start after the specified delay following the completion of the previous execution.

This ensures that there is a gap of at least the specified delay between the end of one execution and the start of the next one.

If a task takes longer to execute than the specified delay, the next execution will be scheduled to start immediately after the previous one completes, ensuring a minimum gap between executions.

**Here's a comparison between the two attributes:**

Use fixedRate when:

You want to execute a task at a consistent rate, regardless of the task execution time.

You want to maintain a steady rhythm for the execution of tasks, even if tasks occasionally take longer to execute.

Use fixedDelay when:

You want to ensure that there is a minimum delay between the completion of one task and the start of the next one.

You want to prevent overlapping executions, ensuring that the next task starts only after the previous one completes, regardless of how long the task takes to execute.

The fixed rate will not wait until the previous method is completed or not. It will get executed at any cost when it is scheduled whereas the fixed Delay ensures that there will be a delay between two methods. The second method will execute only after the first method is executed.

A diagram of a method

Description automatically generated

**Cron Expression**

A cron expression is a string representing a schedule, which is used to configure the timing of recurring tasks in software applications. It's a widely used format for specifying schedules, particularly in Unix-like operating systems and software frameworks like Spring Boot.

There will be 6 digits (\*) in the cron expression. The 7th one is optional i.e. Every year

Example

\* \* \* \* \* \*

Every second,Every minute,Every hour,Every day of the month,Every month,Every day of the week

**Index**

\* denotes Every

? denotes No specific value.

\*/10 denotes per 10 (seconds,minutes,hour)

7-9 denotes Range from 7 to 9

5,6 denotes at 5 and 6

Examples:

1. 0 0 \* \* \* \* - Top of Every hour of every day. (At the 0th second,At the 0th minute of every hour,every day)
2. \*/10 \* \* \* \* \* - For every 10 seconds,it should execute.
3. 0 0 8-10 \* \* \* - 8,9,10 o clock of every day
4. 0 0 6,19 \* \* \* - At 6am,7pm of every day of the month,every month,every day of the week.
5. 0 0/30 8-10 \* \* \* - Every 30 mins of that hour 8-10 at 0 th second ie 8.0,8.30,9.0,9.30
6. 0 0 9-17 \* \* MON-FRI - At 9 am to 5 pm,every hour ,every day,every month,Monday to Friday
7. 0 0 0 25 12 ? - 25th of December Month at 12 pm (midnight)

**Demo Project was done to illustrate Scheduling in Springboot**

The statement will be executed for every 4 seconds because the fixedDelay will add the additional 1000 milliseconds of the thread.

@Scheduled (fixedDelay = 3000)

public void scheduler () throws InterruptedException

{

LocalDateTime current = LocalDateTime.now();

DateTimeFormatter format = DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm:ss");

String formattedDateTime = current.format(format);

log.info ("Scheduler time " + formattedDateTime);

Thread.sleep(1000);

}

The statement will execute for every 3 seconds, and it does not wait for the 1000 milliseconds of the thread which is specified in the method, and it executes after one second initially.

@Scheduled (fixedRate = 3000, initialDelay=1000)

public void scheduler () throws InterruptedException {

LocalDateTime current = LocalDateTime.now();

DateTimeFormatter format = DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm:ss");

String formattedDateTime = current.format(format);

log.info ("Scheduler time " + formattedDateTime);

Thread.sleep(1000);

}

fixedRateString = "PT02S": This parameter specifies the rate at which the annotated method will be executed. Here, it is set to run every 2 seconds. The value is specified in ISO-8601 duration format. "PT02S" stands for a period of 2 seconds.

If we give as PT02H, then it will execute for every 2 hour.

If we give as PT02M, then it will execute for every 2 minutes.

@Scheduled (fixedRateString = "PT02S”, initialDelay=1000)

public void scheduler () throws InterruptedException {

LocalDateTime current = LocalDateTime.now();

DateTimeFormatter format = DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm:ss");

String formattedDateTime = current.format(format);

log.info ("Scheduler time " + formattedDateTime);

Thread.sleep(1000);

}

@Async

@Scheduled (cron = "\*/2 \* \* \* \* \* ")

The scheduling is done through the cron expression here. The statement will execute for every 2 seonds and the annotation is used for parallel execution. @EnableAsync annotation is given to the whole class.

public void scheduler () throws InterruptedException

{

LocalDateTime current = LocalDateTime.now();

DateTimeFormatter format = DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm:ss");

String formattedDateTime = current.format(format);

log.info ("Scheduler time " + formattedDateTime);

Thread.sleep(1000);

}

@Scheduled (cron= "0 \*/2 \* \* \* \*") //For every 2 minutes it should execute

public void scheduler () throws InterruptedException {

LocalDateTime current = LocalDateTime.now();

DateTimeFormatter format = DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm:ss");

String formattedDateTime = current.format(format);

log.info ("Scheduler time " + formattedDateTime);

Thread.sleep(1000);

}

@Scheduled (cron= "0 10 20 \* \* TUE") //On Tuesday, At 8:10 pm it should execute

public void scheduler () throws InterruptedException {

LocalDateTime current = LocalDateTime.now();

DateTimeFormatter format = DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm:ss");

String formattedDateTime = current.format(format);

log.info ("Scheduler time " + formattedDateTime);

Thread.sleep(1000);

}

@Scheduled (cron="${cron. expression.value}") //From App Properties

public void scheduler () throws InterruptedException {

LocalDateTime current = LocalDateTime.now();

DateTimeFormatter format = DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm:ss");

String formattedDateTime = current.format(format);

log.info ("Scheduler time " + formattedDateTime);

Thread.sleep(1000);

}

The parallel execution can also be done through ThreadPoolTaskScheduler where we must set the number of thread size instead of @EnableAsync and @Async

@Bean

public ThreadPoolTaskScheduler taskScheduler()

{

ThreadPoolTaskScheduler scheduler = new ThreadPoolTaskScheduler();

scheduler.setPoolSize(20); // Set the desired pool size

return scheduler;

}

**Scheduling the email in Springboot**

Demo Project was done to illustrate the scheduling of Email sending

SMTP: SMTP (Simple Mail Transfer Protocol) is a protocol used for sending email messages between servers. It's the standard protocol for sending emails across the internet. SMTP servers handle outgoing mail.

**Dependencies Used:**

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-mail</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

**Application Properties file:**

We are using the SMTP Office365 server for email communication. The typical port of Office365 will be 587.The username and password with which we are sending is specified here.

spring.mail.properties.mail.smtp.auth: Indicates whether SMTP authentication should be enabled (true in this case).

spring.mail.properties.mail.smtp.starttls.enable: Indicates whether TLS should be enabled for SMTP communication (true in this case). STARTTLS is a way to take an existing insecure connection and upgrade it to a secure one using SSL/TLS.

spring.mail.host=smtp.office365.com

spring.mail.port=587

spring.mail.username=no-reply@neokred.tech

spring.mail.password=n$Ku3}\_3Q0U!

spring.mail.properties.mail.smtp.auth=true

spring.mail.properties.mail.smtp.starttls.enable=true

EmailSenderService Class:

The JavaMailSender object is injected in the class. The sendSimpleEmail method is used to set the From Email, To Email,Subject and Body.

@Autowired

private JavaMailSender mailSender;

public void sendSimpleEmail(String toEmail,

String subject,

String body

) {

SimpleMailMessage message = new SimpleMailMessage();

message.setFrom("no-reply@neokred.tech");

message.setTo(toEmail);

message.setText(body);

message.setSubject(subject);

mailSender.send(message);

System.out.println("Mail Sent Sucessfully...");

}

The @EventListener annotation in Spring is used to declare a method as an event listener. It indicates that the annotated method should be invoked when a specific event occurs. In this case, ApplicationReadyEvent is the event being listened for.

When the Spring application context is fully started and initialized, the ApplicationReadyEvent is published. This event indicates that the application is ready to service requests.

The mail will be sent for every one minute and the receiver mail,subject and body is passed here.

@EventListener (ApplicationReadyEvent.class)

@Scheduled (fixedRate = 60000)

public void triggerMail() throws MessagingException {

senderService.sendSimpleEmail("suraj@neokred.tech",

"Testing Scheduler",

"Scheduling in Springboot is done succesfully");

}

**JobScheduling in JobRunr Portal**

JobRunr provides job scheduling capabilities for executing background tasks or jobs at specific times or intervals. Here's an overview of how job scheduling works in JobRunr:

* Define Job Classes: First, you define the job classes that represent the tasks you want to execute. These job classes typically implement the Job interface provided by JobRunr.
* Schedule Jobs: Once you have defined your job classes, you can schedule them for execution using JobRunr's scheduling APIs. You can specify when you want the job to run, either as a one-time execution or as a recurring task.
* Specify Execution Time: When scheduling a job, you can specify the exact time at which you want it to be executed. JobRunr will ensure that the job runs at the specified time.
* Handle Job Execution: JobRunr provides mechanisms for handling the execution of scheduled jobs. When the scheduled time arrives, JobRunr will automatically execute the job by invoking the appropriate method in your job class.
* Monitoring and Management: JobRunr includes a dashboard that allows you to monitor the execution of scheduled jobs, view their status, and manage them as needed. This dashboard provides visibility into the scheduling and execution of background tasks within your application.

Demo Project was done to illustrate the scheduling using JobRunr Dashboard

**Dependencies used:**

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.jobrunr</groupId>

<artifactId>jobrunr-spring-boot-starter</artifactId>

<version>5.3.3</version>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>com.mysql</groupId>

<artifactId>mysql-connector-j</artifactId>

<scope>runtime</scope>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<dependency>

<groupId>com.fasterxml. jackson.core</groupId>

<artifactId>jackson-databind</artifactId>

<version>2.15.2</version>

</dependency>

**Application Properties file**

A database configuration must be done for JobRunr project, and we should enable the jobRunr server and jobRunr dashboard.

spring.datasource.password=root

spring.datasource.url=jdbc: mysql://localhost:3306/jobRunr

spring.datasource.username=root

spring.jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true

org.jobrunr.background-job-server.enabled=true

org.jobrunr.dashboard.enabled=true

SampleService Class:

The doLongRunningTask method is written by passing the name in the parameter.This method is basically written for the enqueuing of the job which is written in controller layer

public void doLongRunningTasks(String name) throws InterruptedException

{

System.out.println("Hello "+name+"from JobRunr");

Thread.sleep(10000);

}

The myLongRunningMethod is written for scheduling the job which is written in controller layer. This method basically prints the set of statements with the delay of 5 seconds between them and is called by the controller method

public void myLongRunningMethod() throws InterruptedException

{

System.out.println("The job is started");

Thread.sleep(5000);

System.out.println("In the middle");

Thread.sleep(5000);

System.out.println("The job is done");

}

The myLongRunningMethod2 is written for scheduling recurring job which is written in controller layer. This method basically prints the set of statements with the delay of 5 seconds between them and is called by the controller method.

public void myLongRunningMethod2() throws InterruptedException

{

System.out.println("The job is started");

Thread.sleep(5000);

System.out.println("In the middle");

Thread.sleep(5000);

System.out.println("The job is done");

}

SampleController Class:

@Autowired

private JobScheduler jobScheduler;

@Autowired

private SampleService sampleService;

This controller method is used to enqueue the job. After we hit the “/ enqueue/one-job” endpoint, In the dashboard, firstly the job will be enqueued and then job will be processed and then the job will be succeeded.

@GetMapping("/enqueue/one-job")

public String enqueueOneJob(@RequestParam(name = "name", defaultValue = "world") String name) {

jobScheduler.enqueue(() -> sampleService.doLongRunningTasks(name));

return "One job Enqueued";

}

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

This controller method schedules the job after one minute. So, after I hit the “schedule/one-job”, the method will not execute until 60 seconds, After the 60 seconds it will execute.

Instant.now().plusSeconds(60) is used to execute the job after certain seconds

Instant.now().plusmillis(60) is used to execute the job after certain milliseconds

Instant.now().plusNanos(60) is used to execute the job after certain nanoseconds

@GetMapping("/schedule/one-job")

public String ScheduleOneJob() {

jobScheduler.schedule(Instant.now().plusSeconds(60),()>sampleService.myLongRunningMethod());

return "One job Scheduled";

}

This method is an example for scheduling the jobs which are recurrently. The method is written for scheduling the job at 12 am at midnight every day. The job will not execute until the present day is over unless we trigger it manually

@GetMapping("/recurring/one-job")

public String ScheduleRecurringJob() {

jobScheduler.scheduleRecurrently(Cron.daily(), ()->sampleService.myLongRunningMethod2());

return "One job is Scheduled Recurrently ";

}

A screenshot of a computer

Description automatically generated