**Simple Job Management System**

Detailed System Design

Table of Contents

[**Purpose of this document:** 3](#_Toc533618802)

[**Overview of the System:** 3](#_Toc533618803)

[**Features:** 3](#_Toc533618804)

[**Assumptions:** 3](#_Toc533618805)

[**Improvements:** 4](#_Toc533618806)

[**Architecture and Flow:** 4](#_Toc533618807)

[**Source Code:** 5](#_Toc533618808)

[**Developing New Jobs:** 5](#_Toc533618809)

[**Creating Executable SJMS jar:** 7](#_Toc533618810)

[**Installation/Running Application:** 7](#_Toc533618811)

[**Adding Jobs:** 9](#_Toc533618812)

[**Update Job:** 12](#_Toc533618813)

[**List All Jobs:** 13](#_Toc533618814)

[**List All Job Activities:** 14](#_Toc533618815)

[**Cancel Job Activity:** 15](#_Toc533618816)

[**Run job instantly/Manually:** 16](#_Toc533618817)

[**ScheduleManager:** 16](#_Toc533618818)

[**Activity Diagram:** 19](#_Toc533618819)

[**Sequence Diagram:** 22](#_Toc533618820)

[**Class Diagram:** 22](#_Toc533618821)

[**Screens in thought:** 23](#_Toc533618822)

[**Database tables:** 26](#_Toc533618823)

[**Tools Used:** 28](#_Toc533618824)

[**Running Application in STS** 28](#_Toc533618825)

## **Purpose of this document:**

This document contains the high level design details of Simple Job Management Service. It includes the architecture of the system and use cases.

## **Overview of the System:**

SJMS is an abbreviation for Simple Job Management system. It provides the necessary API to manage the multiple jobs. Jobs are independent for SJMS system. SJMS not aware of what the configured jobs are doing. Each run of job is called as Job Activity and they are maintained separately so user can get the history of job. Jobs can be updated and job activity can be canceled at any point.

## **Features:**

This simple job management is a REST api where Admin/User can manage their jobs easily. It has the following features.

1. It is flexible, there are no change required to the SJMS to develop the new jobs. Jobs are independent from SJMS. User can schedule the same job with multiple schedules. Only restart of the server is required when new jobs are developed, but no changes are from SJMS code base.
2. It is reliable, each job submitted/scheduled is independent task. Completion/Failure of such job don’t have any impact on the other jobs.
3. State of the job can be tracked through API at any given time. Job state will be for Job Activity. Each run will have its own state. Job state (Each run) can be one of five states **QUEUED**, **RUNNING**, **SUCCESS**, **FAILED or CANCELLED**.

QUEUED -- Queued for running.

RUNNING – Currently job is running.

SUCCESS – Job Successfully completed

FAILED – Failued due to some error.

CANCELLED – Updated the schedule with new/Canceld the job activity.

1. Submission of the jobs based on the priority.
2. Jobs can run immediately or as per scheduled.
3. User can update the jobs with new schedules, and they can cancel the job activity (Already scheduled Job Activity) at any given point of time.

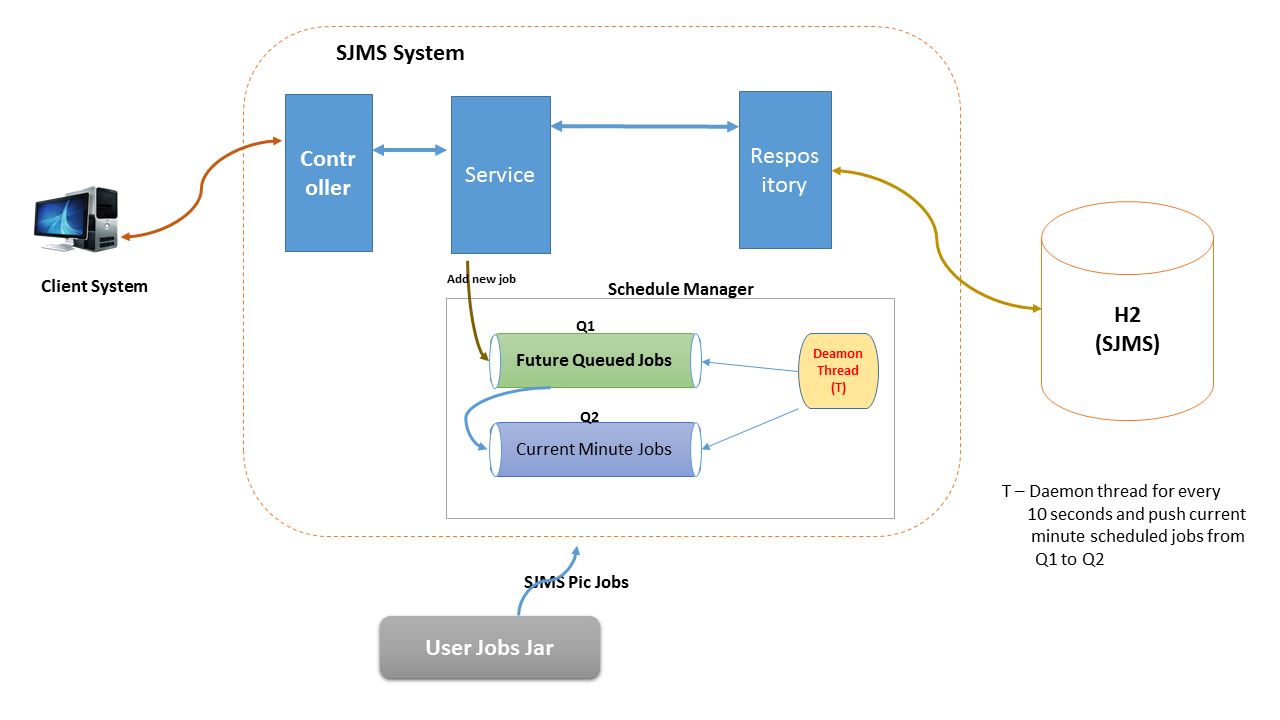
## **Assumptions:**

1. When adding the new jobs, server restart is required which don’t show any impact on the future scheduled jobs.
2. Minimum scheduling unit is a minute.
3. Server restarts will happen in a time where no jobs are running to reduce impact on running threads.

## **Improvements:**

1. Scan the job classes and show it in the Job Class drop down in Add New Job screen.
2. Implement cron like schedules.
3. Development will be faster if we use the third party open source schedulers like quartz schedulers.
4. Holiday calendar can be implemented.

## **Architecture and Flow:**



1. Application is built using Spring Boot and Spring data jpa. No third party schedulers are used.
2. SJMS system follows the MVC architecture. Jobs will be picked from User jobs jar file which is not part of SJMS. Jobs are completely independent of SJMS, only user has to do is they have to extend the given class in sjms-core jar (which is client jar shared with users)
3. Start the server by specifying the location of User Jobs, so that SJMS will pick up the User Jobs and now ready to schedule the jobs.
4. User can save/schedule the job by giving the necessary fields like, jobname, jobClass, effectiveDate, endDate and expression (schedule) etc.
5. User can use the following example expressions to schedule job.
   1. @every 1 months – Every Monthly
   2. @every 3 hours – Every 3 Hours (allowed values 1-23)
   3. @every 1 weeks - Every 1 week
   4. @every 2 years – Every 2 years
   5. @every 1 minutes – Every 1 minute(allowed values 1-59)
6. As soon as the job is saved it will be given for scheduling where next fire time will be calculated and put it in queue for running.
7. A Deamon thread runs every 10 seconds to check whether there are any scheduled jobs at this minute.
8. If it finds any jobs at this minute, add all those jobs to Priority Queue (based on Job priority) to pick those jobs based on those priority. Once they are picked, priority will be maintained by Thread Scheduler (Considering priority only while submitting jobs).
9. While job is running we calculate nextfiretime again to check whether it is expired or we need to run for next instance. If there is a schedule for next time also, we will add jobs to queue. This continues for a job until its endDate reaches. If endDate is null, it will be forever.

## **Source Code:**

Source code contains SJMS, SJMS-Core (client jar source) and jobs (Example Jobs code). Source code and all jar files can be found in **github** repository **https://github.com/raghupulishetti/sjms.git**

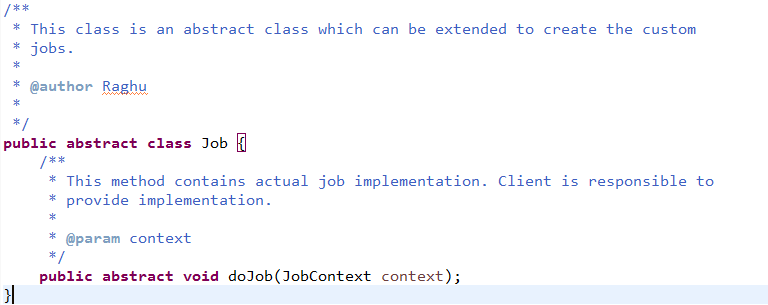


## **Developing New Jobs:**

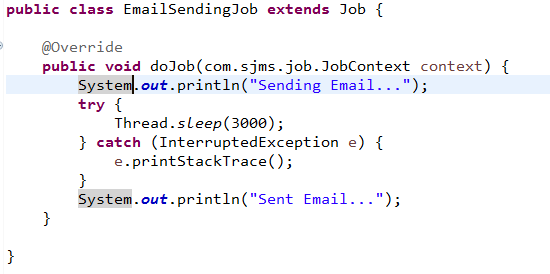
User can develop the custom jobs by extending the **Job** class which is part of client jar (sjms-core.jar). Client (**sjms-core.jar**) jar file can be used to compile the code while developing Jobs. Below is the jar file.



Job class looks like below.



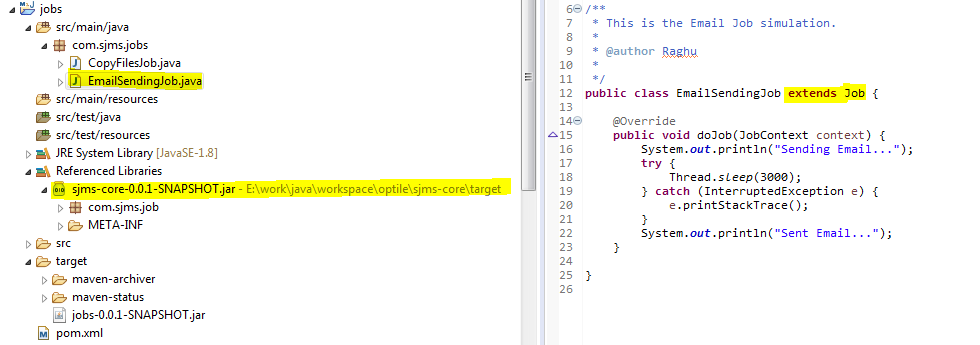
**Example Job:**



**Note:** Please do not include client jar(sjms-core.jar) as it is intended to compile the code.

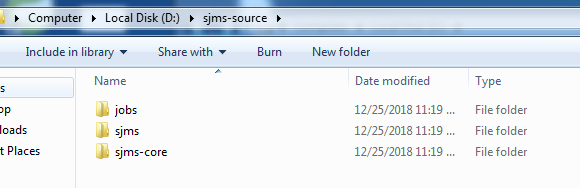
Develop all jobs as shown above and make a jar file say (**user-jobs.jar**).



## **Creating Executable SJMS jar:**

1. Unzip the above source code attached above to a folder.



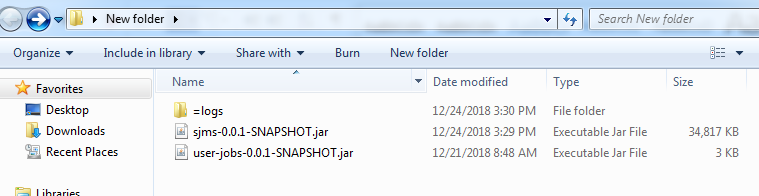
1. Go to the each folder and run mvn clean install command. Make sure you have configured maven, java 1.8. In each project target folder respective jar file will be created.
2. Copy sjms.jar and sjms-core.jar files to a folder. And follow below steps specified in next section.

## **Installation/Running Application:**

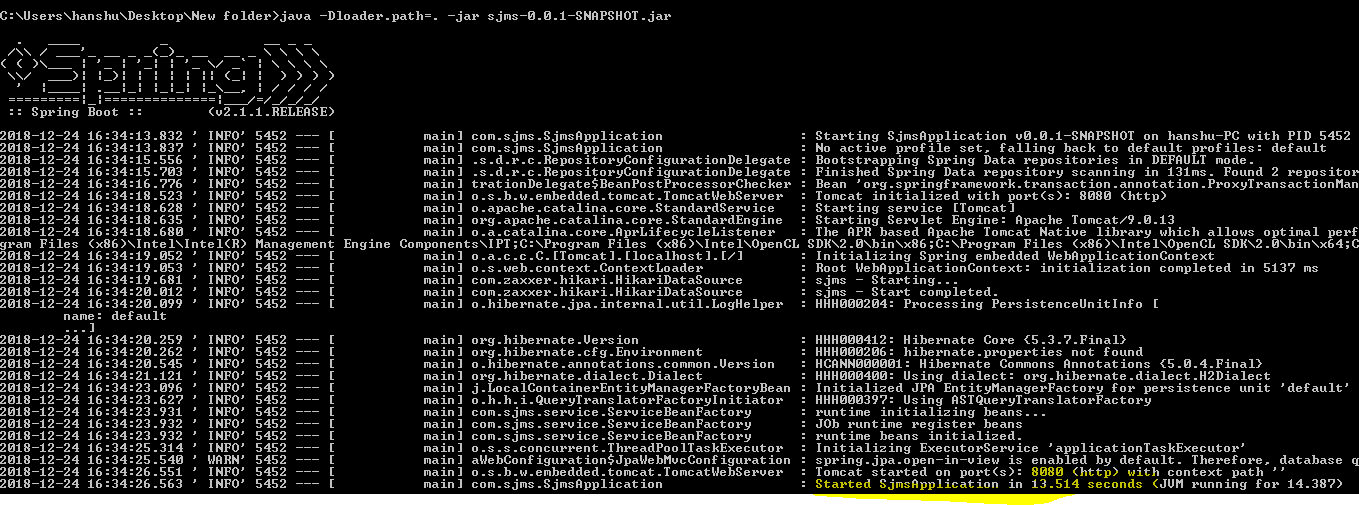
To run SJMS application java 8 mandatory. This application can be run in 2 ways

1. By deploying as war file. – If we deploy as war file, user jobs jar file should be placed in **<Application Name>/WEB-INF/lib** folder and start the server (Tomcat).
2. By running the jar(spring boot has the option to create the executable jar) – In this case we need to have both SJMS jar file and User jobs jar (**user-jobs.jar**) files in same folder and run the following command at same folder location to start the application from the **terminal**. This is simple way of running spring boot applications and I am following this approach to explain the SJMS. Logs related to SJMS can be found in logs folder which will be created in current folder where 2 jar files are places.

**java -Dloader.path=. -jar sjms-0.0.1-SNAPSHOT.jar**



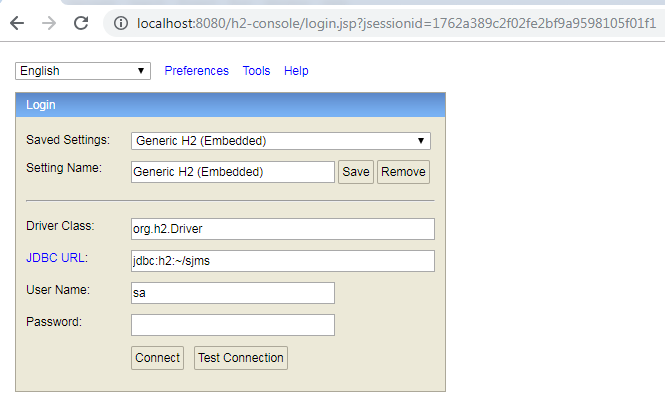




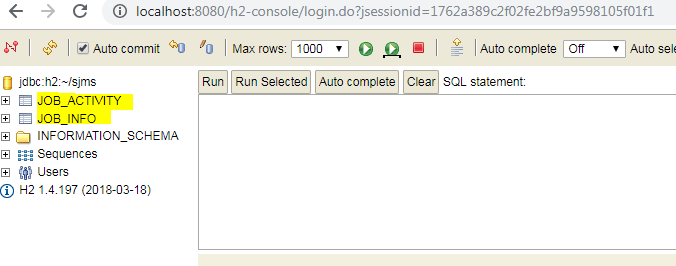
Once the SJMS started screen looks like above.

Database tables will be created automatically once the server is started. We can access the database by using below address from browser and access tables created.

<http://localhost:8080/h2-console/>



Click on connect to see the tables.



## **Adding Jobs:**

After your application has been started, now it’s time to configure user jobs. New jobs can be configured by using rest api provided.

**To add new job**

**API:** <http://localhost:8080/jobs/save>

**Method:** POST

**Request:**

{

"jobName": "Email JOb",

"jobClass": "com.sjms.jobs.EmailSendingJob",

"active": true,

"schedule": "@every 1 minutes",

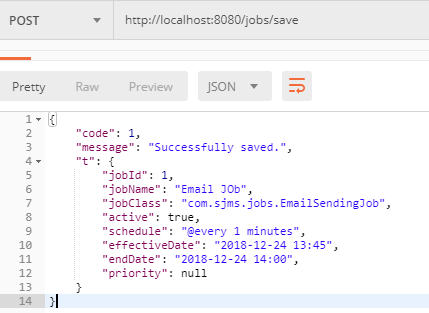
"effectiveDate": "2018-12-24 13:45",

"endDate": "2018-12-24 14:00",

"priority": “HIGH”

}

**Response:**



**Job Configuration Options:**

**jobname** – Can be user choice

**jobClass** – Fully qualified name of Job class.

**active**- true

**schedule** - @every (0-59) minutes or @every (0-23) hours or @every (1-any) weeks or

@every (1-any) months or @every (1-any) years or @onetime

**effectiveDate** – any date

effectiveDate is an optional field

Follow the date format as given in the request if passing.

if we give future date job will start at that time.

if we give the past date, current time will be taken as effective date.

**endDate** – future date

endDate is an optional field

Follow the date format as given in the request if passing.

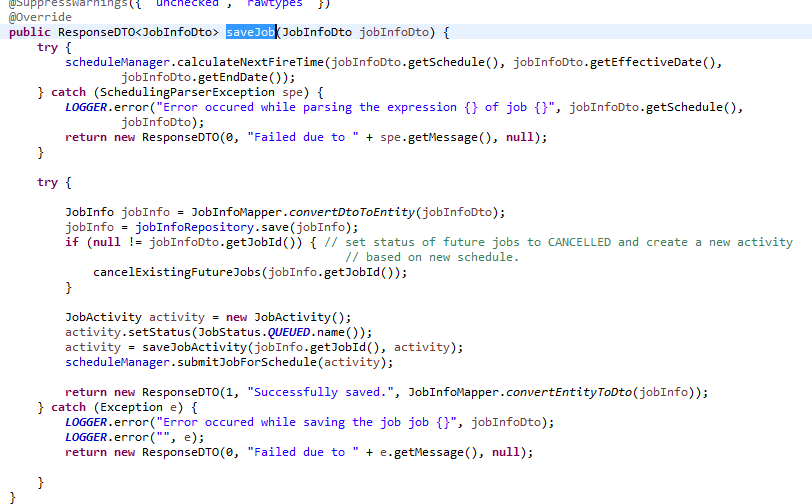
If we give the past date, job will never be triggered.

If we give the future date, job will be trigged based on schedule

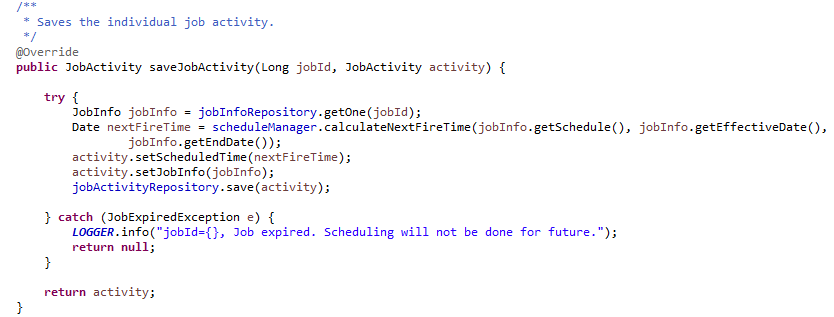
(like for every minute) until endDate reaches. After that it will never trigger.

**priority** – HIGH, MIDIUM or LOW

As soon as we add the new job, request goes to the Job Controller it will dipatch the request to service layer (JobService). If schedule expression and all other inputs are valid then a record will be saved in to “JOB\_INFO” (for each job one record) table.



And a new record will be inserted into the JOB\_ACTIVITY (For each job running instance) table by calculating next fire time based on above configurations.



Once these are saved, job activity will be given for scheduling (added to the queue). SchedulingManager class will take care of scheduling and firing the jobs.

## **Update Job:**

Same save job API will be used for update the Job as well. But in the request along with all the fields, user has to add the extra field “jobId”.

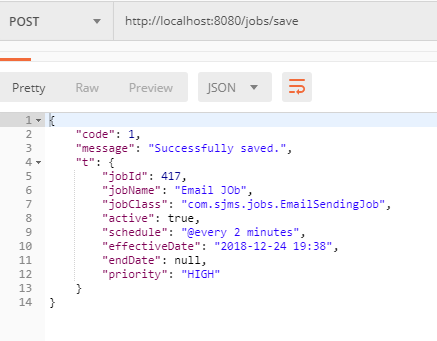
**API:** <http://localhost:8080/jobs/save>

**Method:** POST

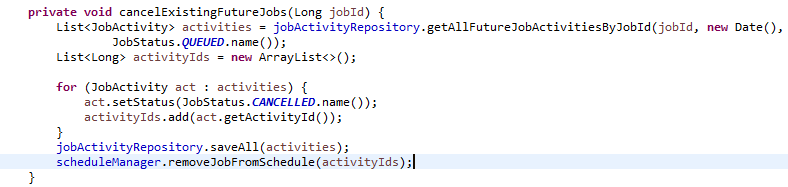
**Request:**



**Response:**



Whenever the user is updating the Job details, updated details will be saved for the given jobId in the JOB\_INFO table. As soon as the details are saved, if there are any scheduled jobs(which are in queue for running) will be canceled (status set to cancelled in JOB\_Activity table) and new record will be inserted into the JOB\_ACTIVITY table with the new schedule given in the update API.



## **List All Jobs:**

We have an API to get the all jobs saved by the USER though REST call. Below are the details of API.

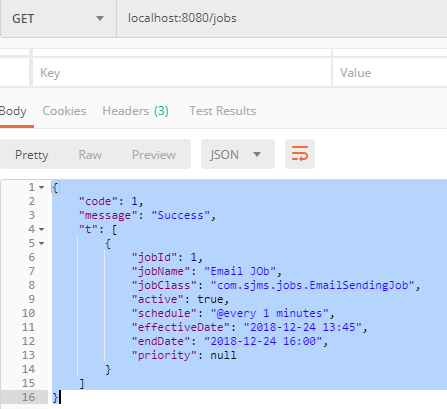
**To get all jobs**

**API:** http://localhost:8080/jobs/

**Method:** GET

**Request:**

**Response:**



## **List All Job Activities:**

This API is used to get all Job instances when it ran, what is the status of the job at that run.

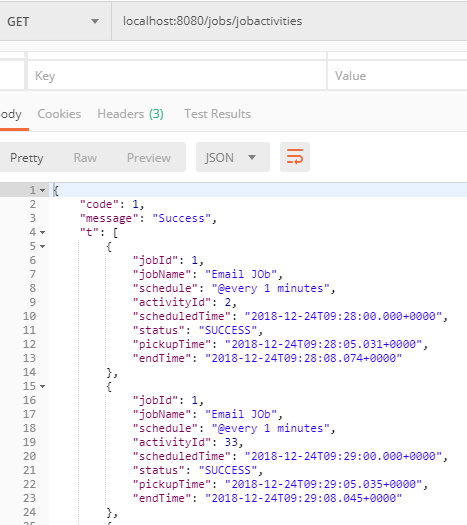
**To get all job activities**

**API:** http://localhost:8080/jobs/jobactivities

**Method:** GET

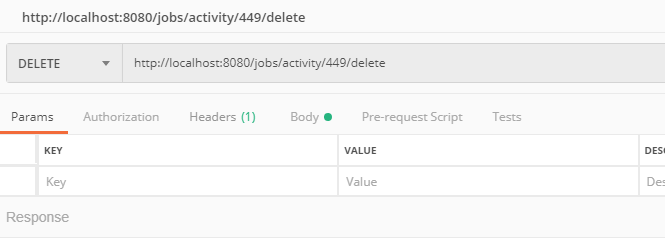
**Request:**

**Response:**



## **Cancel Job Activity:**

User can cancel any QUEUED job instance (future scheduled job activity), so that job activity status will be set to Cancelled and that instance never runs. It will cancel the existing job activity and create new job activity based on the configurations given.



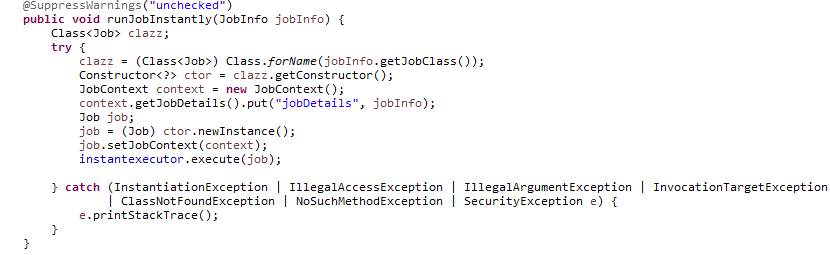
## **Run job instantly/Manually:**

User can run the job instantly irrespective of the schedule. For this user has to hit separate API by giving the job id. Below are the details.

**To run job instatly**

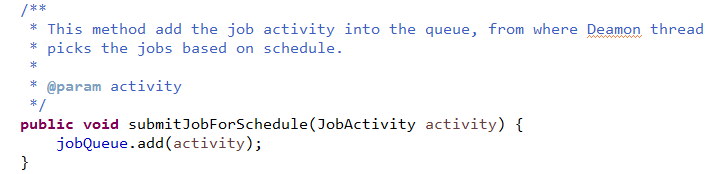
**API:** http:// localhost:8080/jobs/{jobId}/run

**Method:** POST



## **ScheduleManager:**

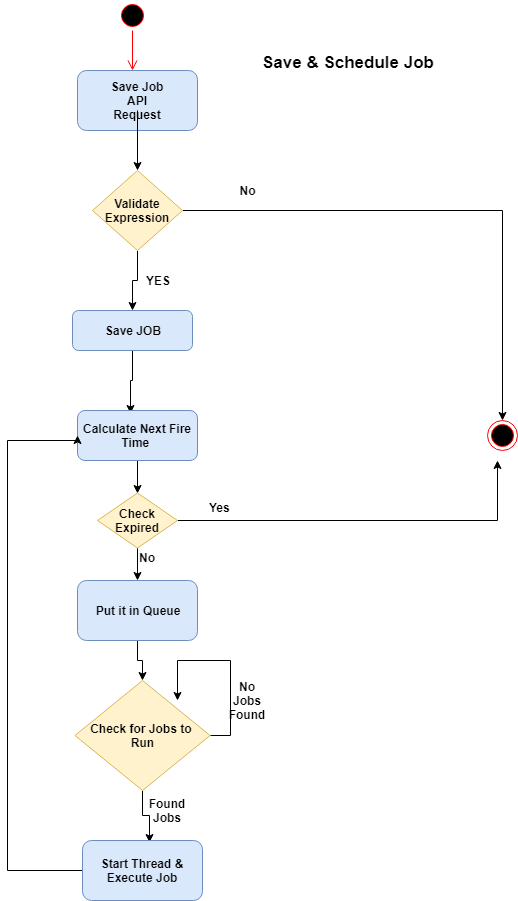
This is the main component of the application where scheduling happens and triggering of the jobs are done. Once the job is saved it will be queued for the next fire.

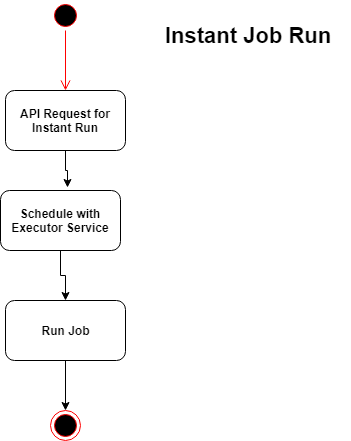


As soon as the ScheduleManager instance is created by Spring container, A daemon thread will be created and it’s responsibility is to continuously monitor the queue and check where there are any threads needs to be triggered at this moment. If there are any jobs to be triggered, those will start running using **ExecutorService**. We are using **java.concurrent** package to avoid problems comes with Threads.

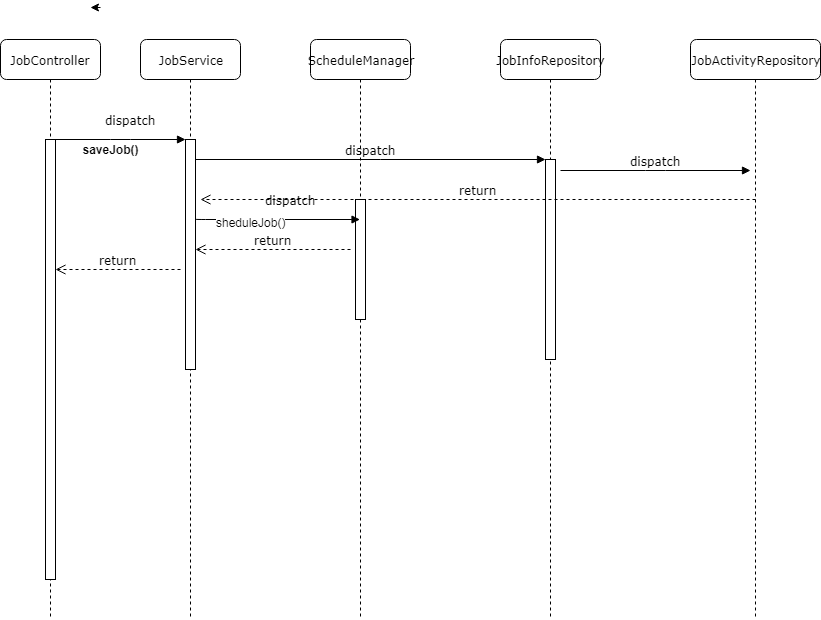
|  |
| --- |
| @PostConstruct  **public** **void** init() {  scheduleFrequency = **new** ArrayList<>();  scheduleFrequency.add("years");  scheduleFrequency.add("months");  scheduleFrequency.add("weeks");  scheduleFrequency.add("days");  scheduleFrequency.add("hours");  scheduleFrequency.add("minutes");  instantexecutor = Executors.*newScheduledThreadPool*(5);  executor = Executors.*newFixedThreadPool*(20);  Thread t = **new** Thread(() -> {  **while** (**true**) {  **try** {  Thread.*yield*();  Thread.*sleep*(10000);  Calendar now = Calendar.*getInstance*();  now.set(Calendar.***SECOND***, 0);  now.set(Calendar.***MILLISECOND***, 0);  Comparator<JobActivity> comp = **new** Comparator<JobActivity>() {  @Override  **public** **int** compare(JobActivity o1, JobActivity o2) {  Integer o1priority = Integer.*valueOf*(o1.getJobInfo().getPriority().getValue());  Integer o2priority = Integer.*valueOf*(o2.getJobInfo().getPriority().getValue());  **return** -o1priority.compareTo(o2priority);  }  };  **final** BlockingQueue<JobActivity> priorityQueue = **new** PriorityBlockingQueue<>(5, comp);  jobQueue.stream().forEach((act) -> {  **if** (act.getScheduledTime().compareTo(now.getTime()) == 0) {  **if** (act.getStatus().equals(JobStatus.***QUEUED***.name())) {  **try** {  priorityQueue.put(act);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  jobQueue.remove(act);  }  });  priorityQueue.stream().sorted(comp).forEach((activity) -> {  Class<Job> clazz;  **try** {  clazz = (Class<Job>) Class.*forName*(activity.getJobInfo().getJobClass());  Constructor<?> ctor = clazz.getConstructor();  JobContext context = **new** JobContext();  context.getJobDetails().put("currentActivity", activity);  context.getJobDetails().put("job", activity.getJobInfo());  Job job = (Job) ctor.newInstance();  job.setJobContext(context);  executor.execute(job);  } **catch** (ClassNotFoundException | NoSuchMethodException | SecurityException  | InstantiationException | IllegalAccessException | IllegalArgumentException  | InvocationTargetException e) {  JobService jobService = ServiceBeanFactory.*getBean*(JobService.**class**);  jobService.updateJobStatus(JobStatus.***FAILED***, **new** Date(), **new** Date(),  activity.getScheduledTime(), activity.getActivityId(), e.toString());  ***LOGGER***.error("Error occured while starting the job...");  ***LOGGER***.error("", e);  }  });  } **catch** (Exception e) {  ***LOGGER***.error("Error occured while running the deamon thread");  ***LOGGER***.error("", e);  }  }  });  t.setDaemon(**true**);  t.setPriority(Thread.***MIN\_PRIORITY***);  t.start();  } |

## **Activity Diagram:**

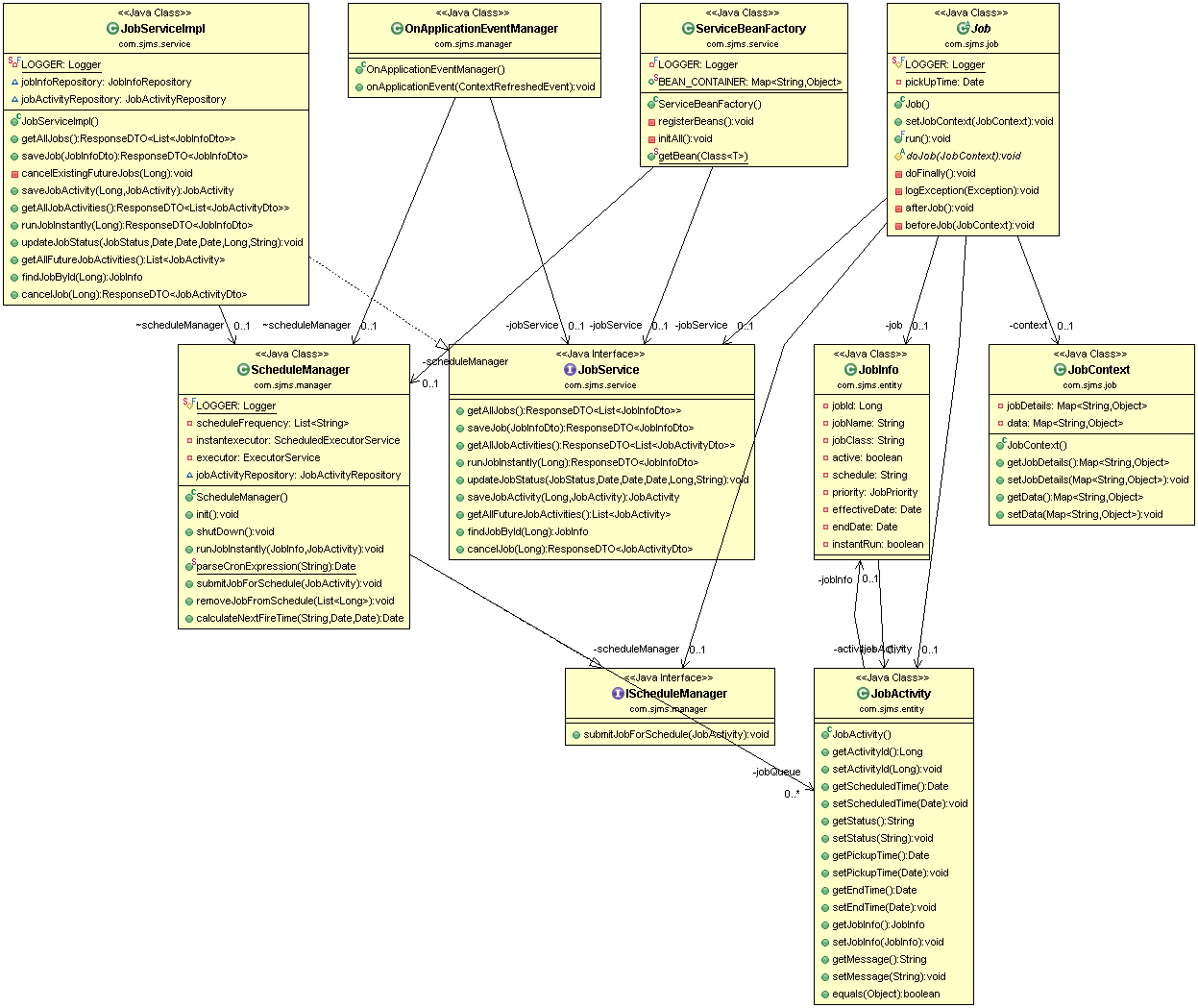




## **Sequence Diagram:**

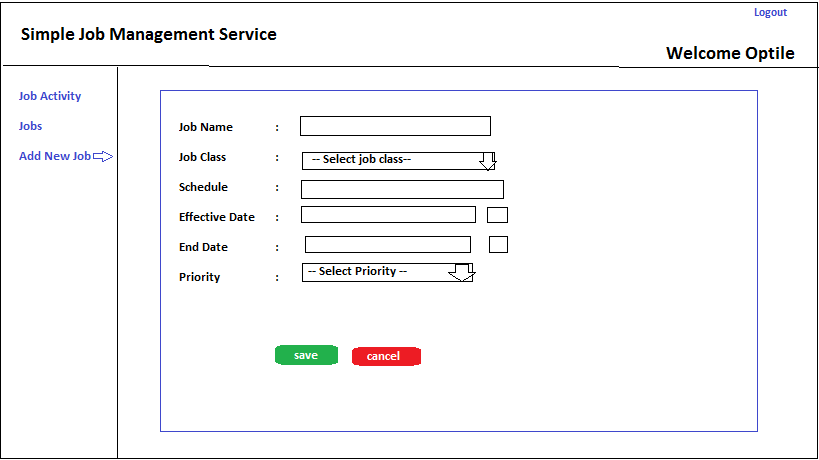


## **Class Diagram:**

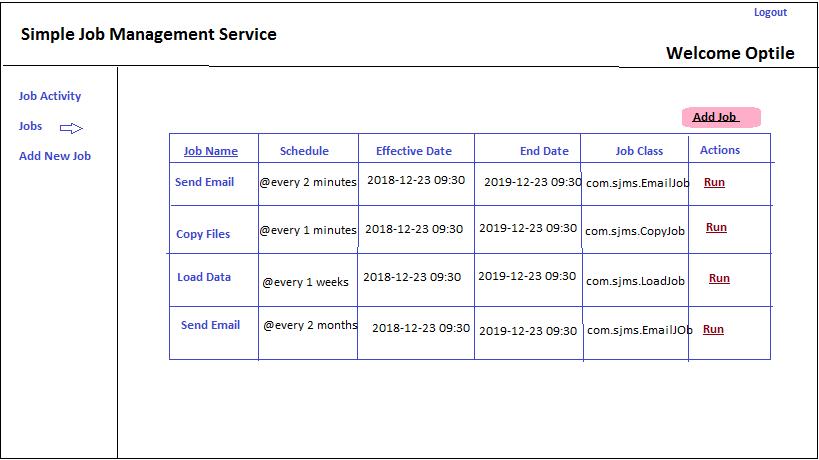


## **Screens in thought:**

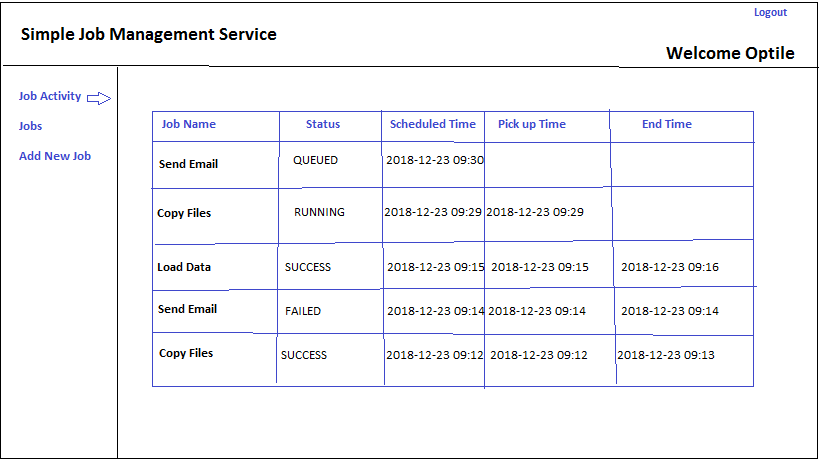
1. **Add a New Job screen** will take the necessary input fields for the JOB. User can select the Job class (this is an improvement, for now user has to give manually) which is scanned while starting the server.
2. On click of save button by giving all the valid inputs new job will be saved and scheduled for running.



1. **Jobs Screen** will show all the jobs as data table which were saved by the user from Add new Job screen. It has Job Name, Schedule, Effective Date, End Date and Job Class as fields.
2. It has Manual Run link which enables the user to run the jobs manually and immediately irrespective of schedule.



1. Job Activity screen contains each run instance. For example if the job is scheduled for every minute, every run of job details will be saved and showed in this.
2. It’s also the data table which contains the fields Job Name, Status, Scheduled Time, Pickup Time and End Time.
3. Status of the job can be in QUEUED, RUNNING, SUCCESS, CANCELLED OR FAILED.

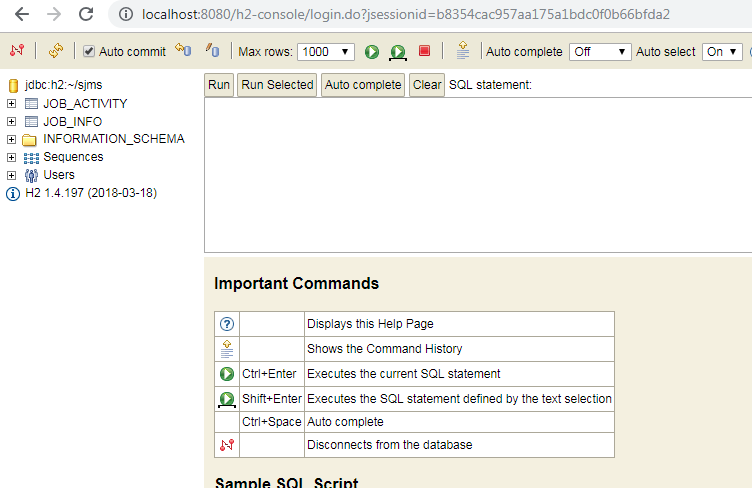


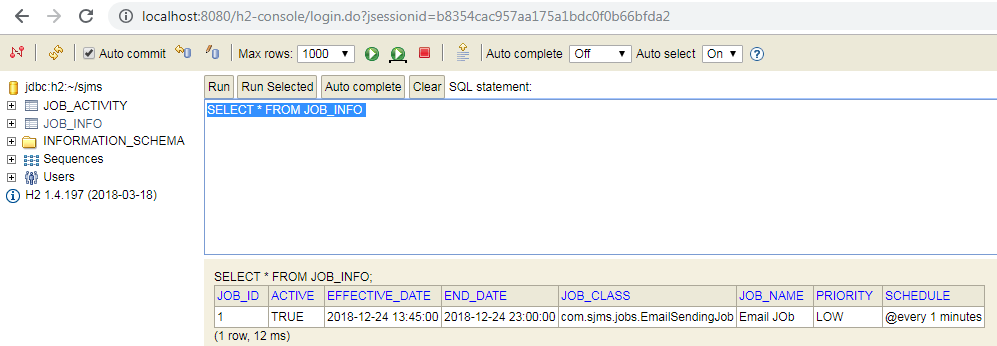
## **Database tables:**

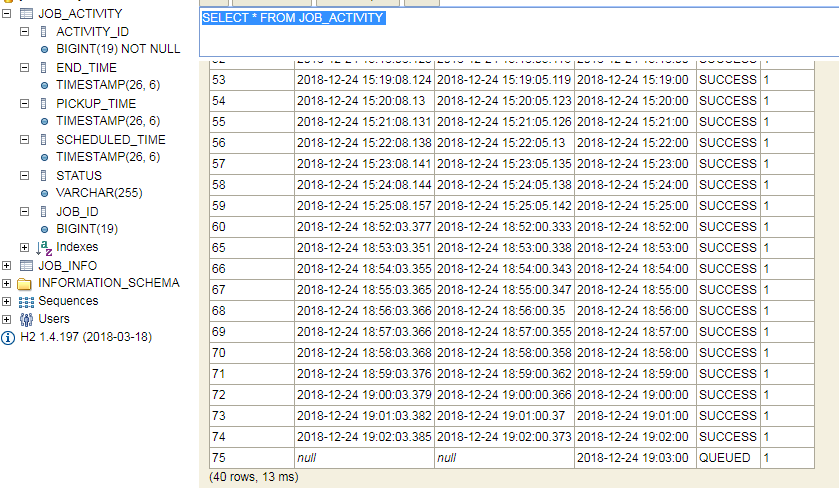
To see H2 database console, after starting the server we need to hit the following address in the browser. As soon as the database is started two tables will be created.

<http://localhost:8080/h2-console/>

1. JOB\_INFO
2. JOB\_ACTIVITY





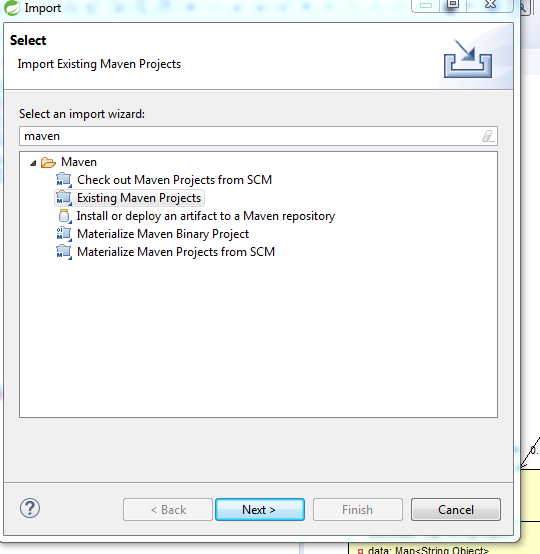


## **Tools Used:**

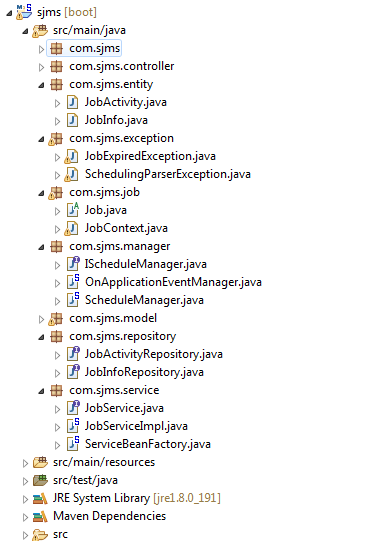
1. Java 1.8
2. STS 3.9.2
3. H2 Database
4. Spring Boot 2.x
5. Maven as a build tool.

## **Running Application in STS**

1. Unzip the source code into a directory.
2. Open the STS/Eclipse
3. Right click in the package explorer and select import as existing maven project.

e

1. Select the unzipped folder SJMS and click on OK.



1. Right click on the project and run as Spring Boot, it will start the embedded tomcat.

