Job Management Case Study

# High-Level Technical Design

Version 1.0

03/25/2018

Table of Contents

[High-Level Technical Design 1](#_Toc509785854)

[1. Purpose of the Document 3](#_Toc509785855)

[2. Intended Audience 3](#_Toc509785856)

[3. Case Study Requirement 3](#_Toc509785857)

[4. High Level Architecture 4](#_Toc509785858)

[5. Implementation Detail 4](#_Toc509785859)

[5.1 Features Covered from Requirement 6](#_Toc509785860)

[6. Assumptions Made for Implementing Prototype 6](#_Toc509785861)

[7. Improvements and Extensions in Prototype 6](#_Toc509785862)

[8. How to Import and Use 7](#_Toc509785863)

[8.1 Reviewing and Testing the Prototype 7](#_Toc509785864)

# Purpose of the Document

Purpose of this document is to provide the technical details of Job Management case study design and implementation.

# Intended Audience

Intended audience will be technical architects and developers who want to use and review the Job Management System.

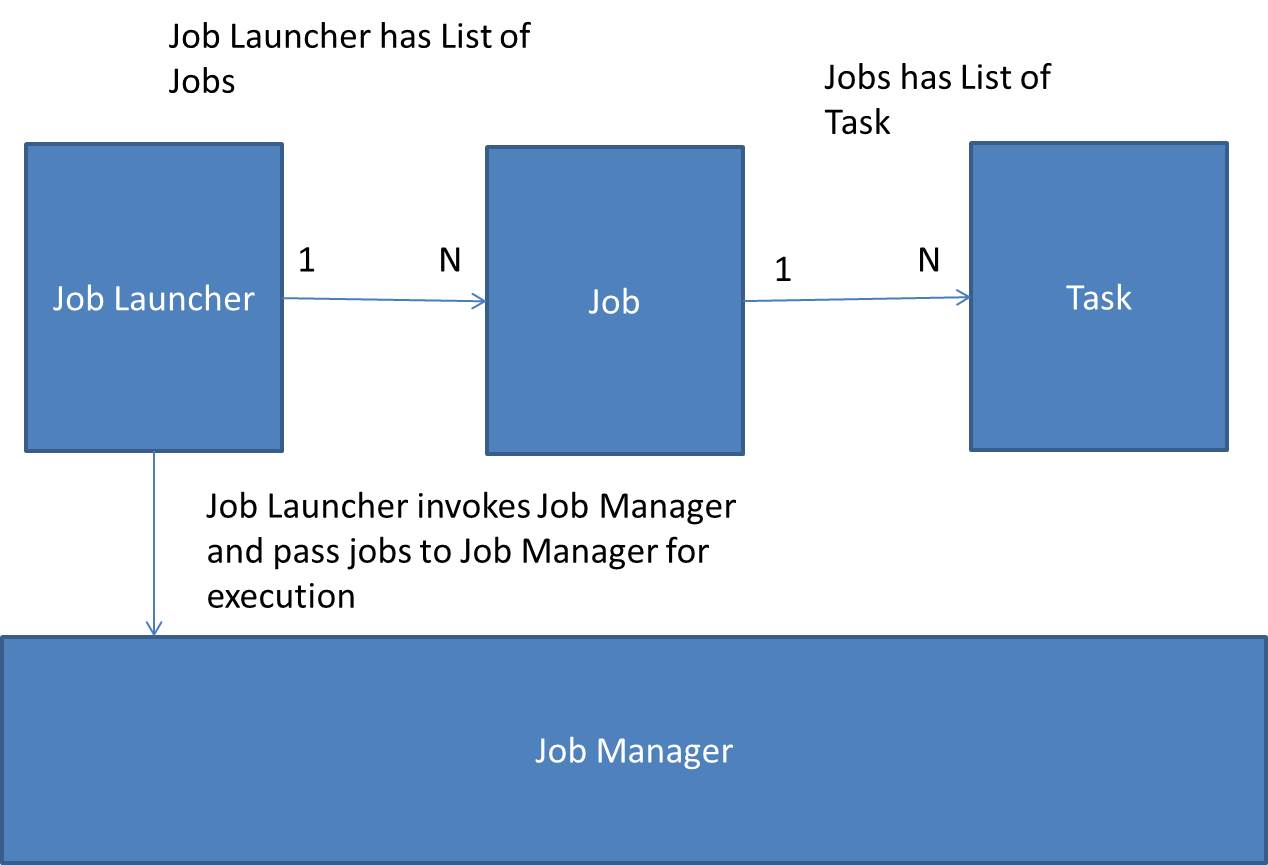
# Case Study Requirement

Requirement of the case study is to implement the Job Management System prototype, which can execute the job without having the knowledge of actions performed by job. Also it should support creation of new jobs without modifying the Job Manager.

Following features should be taken into consideration while designing the prototype.

1. **Flexibility** - The types of possible actions performed by the Jobs are not known to the Job Management System. In the future, new Jobs should be supported without re-developing the Job Management System (optional).
2. **Reliability** - Each Job should either complete successfully or perform no action at all. (I.e. there should be no side-effects created by a Job that fails.)
3. **Internal Consistency** - At any one time a Job has one of four states: ***QUEUED*, *RUNNING*, *SUCCESS*, *FAILED***. Following the execution of a Job, it should be left in an appropriate state.
4. **Priority (Optional)** -Each Job can be executed based on its priority relative to other Jobs
5. **Scheduling** - A Job can be executed immediately or according to a schedule.

# High Level Architecture



# Implementation Detail

Job Manager Prototype is completely implemented in core java it neither depends on any framework (like spring) nor on any JEE component. Except Logback and Junit no other third party dependency is used. Builder, Singleton and Observer design patterns are used in implementation and Maven is used to build the prototype. This prototype is reusable component, developer just needs to add it in dependency and start using it.

Below is the abstract detail of Main classes (for complete details check javadocs)

1. **Job** – This is the main interface which provides methods for execution of List of tasks.
2. **AbstractJob** – This class implements Job interface and provide implementation to all the generic methods required to execute the List of Tasks. It also implements Comparable interface to provide the provision of setting priority of Job.
3. **JobIml** – This class extends AbstractJob and set the required property of Job for execution.
4. **Task** – This interface defines the method which will be executed as a part of job execution. This interface should be implemented by developer who is going to use Job Manager and set the List of task in Job.
5. **JobBuilder** – This class is provided to build the Job. For developer they just need to create instance of JobBuilder set the Job properties and job creation is done. Developer doesn’t need to extend AbstractJob or Implement Job Interface. Still if developer wants to provide his/her own implementation then developer is free to do this by extending AbstractJob class.
6. **JobManger** – This interface defines the method to execute the Job. JobManager is not aware what action job is going to perform. It will take List of Job and execute them.
7. **JonManagerImpl** – This is the implementation of JobManager.
8. **JobQueue** – This is wrapper on PriorityBlockingQueue. If priority is set for the job then it will take the job as per the given priority first for execution. BlockingQueue is used here to avoid any ConcurrentModificationExecption in case of multithreaded environment.
9. **ScheduledJobExecuter** – This class is used to execute scheduled job. To execute scheduled job ScheduleThreadPool is used. It will execute the job as per set schedule.
10. **JobEventListener** – This is the interface listens the events of job, in prototype only one **JobStatusEventListener** has been implemented to listens the status update event. Whenever job status changes it logs the updated event.
11. **Priority** – This is the enum for priority only three priorities are defined HIGH, MEDIUM and LOW.
12. **JobStatus** – This is the enum for status as per requirement four status are provided *QUEUED*, *RUNNING*, *SUCCESS*, *FAILED.*
13. **JobSchedule** – This is the class for setting the Job schedule. Developer can set the Hour and Minute of execution.
14. **JobLauncher** – This interface provides the method which adds jobs to JobQueue and invokes the JobManager.
15. **JobLauncherImpl** – This is the implementation of JobLauncher.

## Features Covered from Requirement

There are six features which should be covered as a part of prototype implementation.

1. **Flexibility** – JobManager is flexible enough, it just need any job which implements Job interface. It will execute it.
2. **Reliability** – This is also taken care in prototype. If any job fails it won’t affect the execution of other jobs and Job will not leave in any inconsistent state either it is failed or success once it start execution.
3. **Internal Consistency** – Below are the events when job status changes.
   1. **QUEUED –** When job is added to job queue.
   2. **RUNNING -** When job starts execution.
   3. **SUCCESS -** When job completed its execution successfully.
   4. **FAILED** - When any exception encountered while execution of job.

Job will never be left in inconsistent state.

1. **Priority (Optional)** – Developer can set the priority of Job. If priority is not set then job will be executed as per the order they have added in queue.
2. **Scheduling** – Developer can set the schedule for the job. If schedule is not set then job will be executed immediately.

# Assumptions Made for Implementing Prototype

Following assumptions made for implementing prototype.

1. I have not used any framework or JEE component to implement the prototype, it is completely implemented in core java. This makes the prototype independent of any framework. **Still if you want this prototype to be implemented using Spring or Spring boot then just let me know I will migrate the same.**
2. I have implemented this prototype in Java 8 but not used the new features like optional, streams, lamda etc. Only DateTime api is used for scheduling purpose.
3. As this is the prototype I have not implemented any DB interaction. S**till if you want me to use DB then I can use In-memory DB along with JPA in prototype.**
4. For scheduling a job I have used ScheduledThreadPool and provide only two options to set hour of day and minute of hour. We can also use Quartz scheduler which provides full flexibility.

# Improvements and Extensions in Prototype

1. This prototype can be extended to provide Task level listener.
2. For job scheduling Quartz can be used.
3. We can implement JPA for maintaining job status in DB.

# How to Import and Use

1. Download the project from Github

<https://github.com/ankitshrivastava958/jobmanager.git>

1. Import it as a maven project.
2. Run clean install maven command.
3. Copy the jar (jobmanager-0.0.1-Snapshot.jar) from target folder and add it in your library.
4. Create the Instance of JobBuilder and set the Job properties and get the Job instance.
5. Create instance of JobLauncher, call the launchJobs method and pass the list of Job to it.

## Reviewing and Testing the Prototype

1. Download the project from Github

<https://github.com/ankitshrivastava958/jobmanager.git>

1. Import it as Maven project.
2. Run clean install and then eclipse clean.
3. JobManagerTest class is provided to test the prototype. Inside test class 3 jobs are created to test the different scenario.
4. Developer can also create Job and execute it, below is the sample code.

**//Create Job from JobBuilder**

Job readJob = new JobBuilder()

.setJobName("ReadingJob")

.setJobPriority(Priority.HIGH)

.setJobTasks(getTaskList("I am Reading File")).build();

Job writeJob = new JobBuilder()

.setJobName("WritingJob")

.setJobTasks(getTaskList("I am Writing File"))

.setJobPriority(Priority.LOW).build();

Job deleteJob = new JobBuilder()

.setJobName("DeletingJob")

.setJobPriority(Priority.MEDIUM)

.setJobTasks(getTaskList("I am Deleting File")).build();

List<Job> jobs = new ArrayList<Job>();

jobs.add(readJob);

jobs.add(writeJob);

jobs.add(deleteJob);

**//Create JobLauncher and call launchJobs**

JobLauncher launcher = JobLauncherImpl.getInstance();

launcher.launchJobs(jobs);