TOOLS

Introducing JobRunr: A distributed job scheduler for Java

A quick tutorial on an open source library that leverages lambdas and Spring

by Guest Author

January 15, 2021

JobRunr is a library that lets you schedule background jobs using a Java 8 lambda. You can use any existing method of Spring services to create a job without the need to implement an interface. A job can be a short- or long-running process, and it will be automatically offloaded to a background thread so that the current web request is not blocked.
To do its job, JobRunr analyzes the Java 8 lambda. It serializes the lambda as JSON and stores it in your choice of a relational database or a NoSQL datastore.

If you see that JobRunr is producing too many background jobs and your server cannot cope with the load, you can easily scale horizontally just by adding extra instances of the application. JobRunr will share the load automatically and distribute all jobs over the different instances of your application.

JobRunr also contains an automatic retry feature with an exponential back-off policy for failed jobs. There is also a built-in dashboard that allows you to monitor all jobs. JobRunr is self-maintaining: Successful jobs are automatically deleted after a configurable amount of time, so there is no need to perform manual storage cleanup.

You can find JobRunr on GitHub. I've also uploaded the source for the example below on GitHub. The library is free for commercial use.

Setting up JobRunr

Maven dependency. You need to have the following Maven dependency declared in your `pom.xml` file:

```
<dependency>
    <groupId>org.jobrunr</groupId>
    <artifactId>jobrunr-spring-boot-starter</
    <version>1.2.0</version>
</dependency>
```

Adding required properties. Because you're using the `jobrunr-spring-boot-starter` dependency, the rest of the setup process is easy. Simply add these properties to `application.properties`:

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

The first property tells JobRunr to start an instance of a `BackgroundJobServer` that is responsible for processing jobs. The second property tells JobRunr to start the embedded dashboard. More documentation is available on jobrunr.io.

By default, the `jobrunr-spring-boot-starter` will try to use your existing `DataSource` in the case of a relational database to store all the job-related information. However, since you will use an in-memory datastore, you need to provide a `StorageProvider` bean. The `JobMapper` bean will be provided by the following:

```
jobrunr-spring-boot-starter):java @Bean publi
```

Using JobRunr

Injecting dependencies. To create jobs, inject the jobScheduler and your existing service from which you want to create a job:

`` ```java @Inject private JobScheduler jobScheduler; ``
`` @Inject private SampleJobService sampleJobService; ``` ``

Creating fire-and-forget jobs. Now that you have the dependencies injected, you can create fire-and-forget jobs using the `enqueue` method:

`jobScheduler.enqueue(() -> sampleJobService.executeSampleJob());`

Jobs can have parameters, just like any other lambda, for example:

```
jobScheduler.enqueue(() ->
sampleJobService.executeSampleJob("some
string"));
```

What's going on behind the scenes? JobRunr takes the lambda and analyzes it using the ASM Java bytecode manipulation and analysis framework. It extracts the correct class (in this case, `SampleJobService`) and the correct method (`executeSampleJob`) and serializes all this information together with the parameters into a small JSON object:

```
{
  "lambdaType": "org.jobrunr.jobs.lambdas.Job
  "className": "com.example.services.SampleJo
  "methodName": "executeSampleJob",
  "jobParameters": [
    {
      "className": "java.lang.String",
      "object": "some string"
    }
  ]
}
```

This information, together with some extra information about the job itself, is all serialized via the `StorageProvider` to your choice of datastore (such as a SQL or NoSQL database). One or more `BackgroundJobServers` monitor the `StorageProvider` and take jobs from it.

Since the `BackgroundJobServers` use optimistic locking, each job will be processed only once and all the `BackgroundJobServers` will share the load. This works out great on Kubernetes, where you can scale horizontally to have all jobs processed faster by just bringing up more instances of your application.

Scheduling jobs in the future. You can schedule future jobs with the `schedule` method:

```
jobScheduler.schedule(() ->
sampleJobService.executeSampleJob(),
LocalDateTime.now().plusHours(5));
```

Scheduling recurring jobs. If you want recurrent jobs, use the `scheduleRecurrently` method:

```
jobScheduler.scheduleRecurrently(() ->
sampleJobService.executeSampleJob(),
Cron.hourly());
```

Annotating with @Job. To control all aspects of a job, annotate the service method with the `@Job` annotation, as shown below. This allows you to set the display name in the dashboard and configure the number of retries in case a job fails.

```
@Job(name = "The sample job with variable %0"
public void executeSampleJob(String variable)
    ...
}
```

You can even use variables that are passed to the job in the display name by means of the `String.format()` syntax. If you have very specific use cases where it’s necessary to retry a specific job only upon a certain exception, you can write an `ElectStateFilter` that has access to the job and full control over how to proceed.

The JobRunr dashboard

JobRunr comes with a built-in dashboard that allows you to monitor jobs. If you visit http://localhost:8000, you can inspect all the jobs, including recurrent jobs, as shown in **Figure 1**.

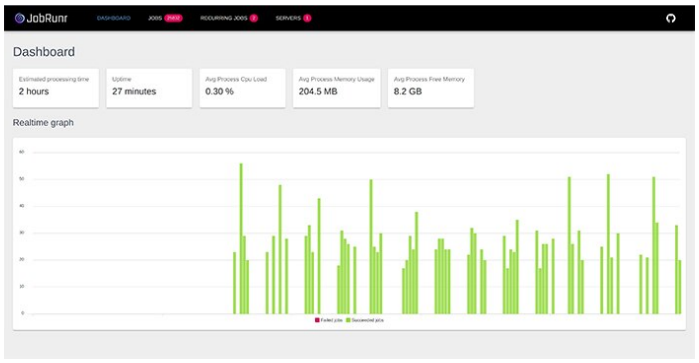
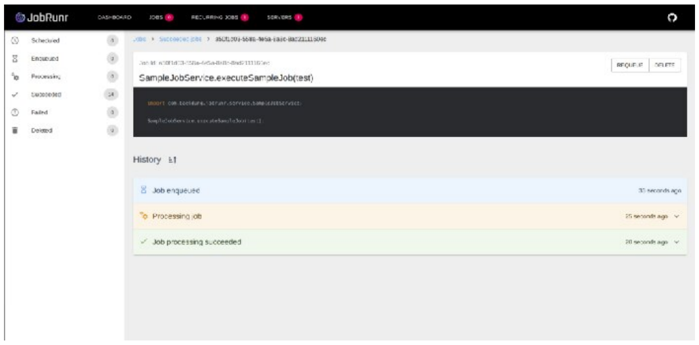


Figure 1. The JobRunr dashboard

Bad things sometimes happen. Maybe an SSL certificate expired or a disk is full. JobRunr, by default, will reschedule the background job with an exponential back-off policy. If the background job continues to fail 10 times, only then will the job go to the Failed state.

You can then decide whether to requeue the failed job from the dashboard when the root cause has been solved. All of this is visible in the dashboard, including each retry with the exact error message and the complete stack trace of why a job failed, as shown in **Figure 2**.



SampleJobService.executeSampleJob(test)

Figure 2. Details about a job

Conclusion

In this article, I described how to build a basic scheduler using JobRunr with the `jobrunr-spring-boot-starter`. The key takeaway from this tutorial is that you can create a job with just one line of code and without any XML-based configuration or the need to implement an interface.

The complete source code for the example is available on GitHub.

Dig deeper

- The JobRunr library
- Sample code used in this article
- Java tutorials: Lambda expressions
- Behind the scenes: How do lambda expressions really work in Java?

Guest Author