Batch Processing Large Data Sets With Spring Bo and Spring Batch

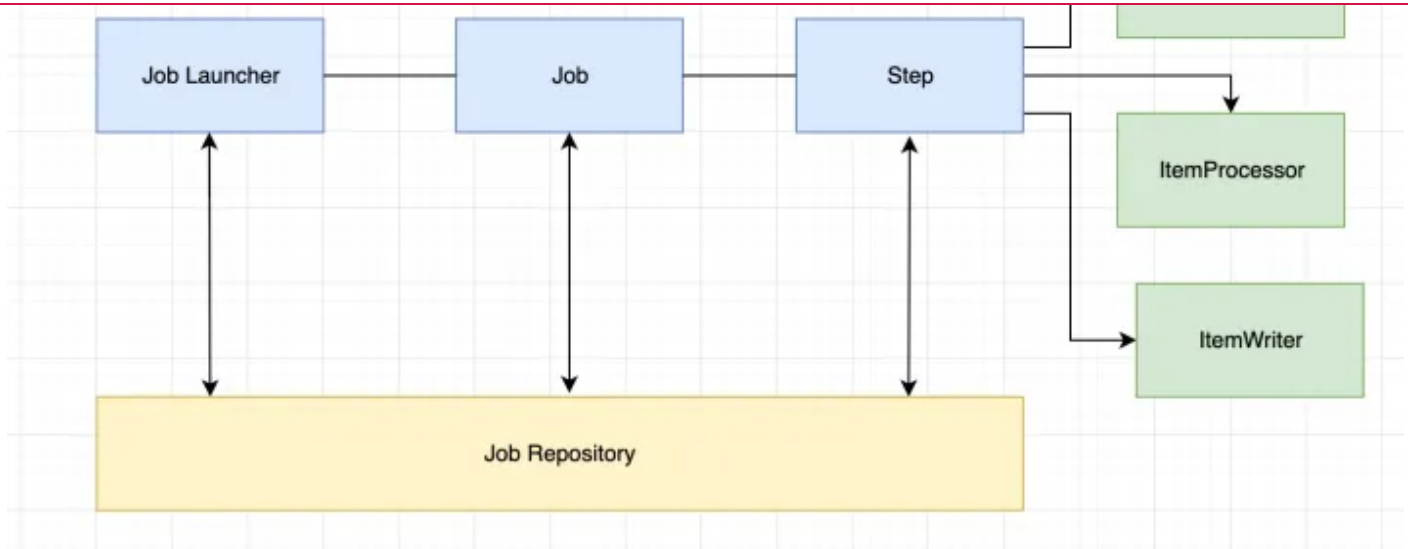
Let's process some data.

by Swathi Prasad MVB · **Aug. 02, 19 · Performance Zone · Tutorial**

Batch processing of data is an efficient way of processing large volumes of data where data is collected, processed and then b results are produced. Batch processing can be applied in many use cases. One common use case of batch processing is transf large set of flat, CSV or JSON files into a structured format that is ready for further processing.

In this article, I am going to demonstrate batch processing using one of the projects of Spring which is Spring Batch. Spring Ba provides functions for processing large volumes of data in batch jobs. This includes logging, transaction management, job res job is not completed), job skip, job processing statistics, and resource management.

Let us look at how Spring Batch works in a nutshell.



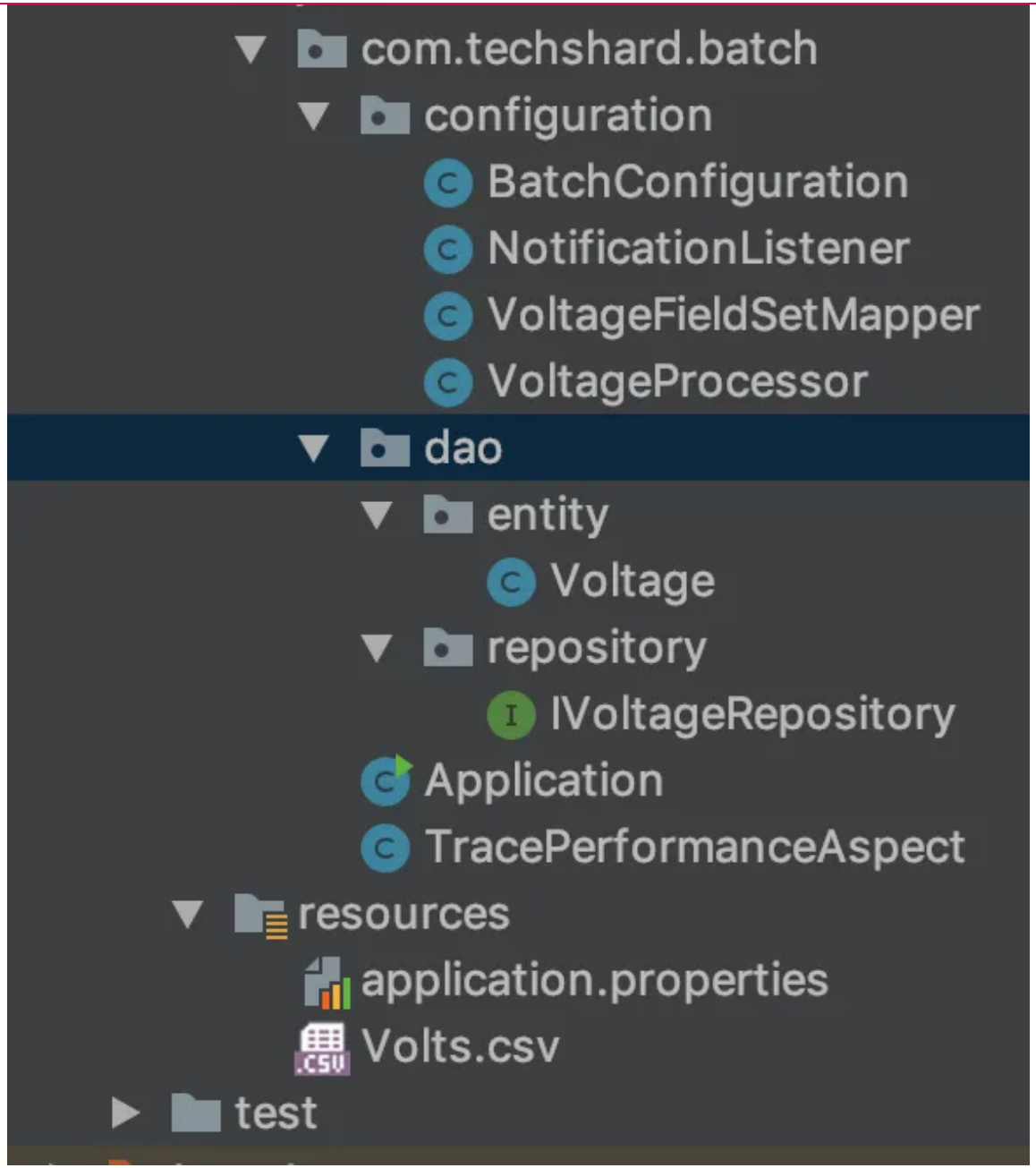
A *step* is an object that encapsulates a sequential phase of a job and holds all the necessary information to define and control processing. It delegates all the information to a Job to carry out its task.

Spring Batch uses chunk oriented style of processing which is reading data one at a time, and creating *chunks* that will be written out within a transaction. The item is read by *ItemReader* and passed onto *ItemProcessor,* then it is written out by *ItemWriter* once the item is ready. The Job Repository will be used to store the step execution periodically during the item processing.

Let's get into coding.

Setting Up the Project

Create a sample Spring Boot application. Here is my sample project structure.



In this article, I will be using sample data which represents voltage drop for a discharging Capacitor. We will read this data from a CSV file and write it out to an in-memory database which is H2.

Add the required dependencies to *pom.xml.*

```
<dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-aop</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-batch</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <dependency>
            <groupId>com.h2database</groupId>
            <artifactId>h2</artifactId>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-api</artifactId>
        </dependency>
```

```
package com.techshard.batch.dao.entity;

import javax.persistence.*;
import javax.validation.constraints.NotNull;
import java.math.BigDecimal;

@Entity
public class Voltage {

    @Id
    @Column (name = "ID", nullable = false)
    @GeneratedValue (strategy = GenerationType.IDENTITY)
    private long id;

    @NotNull
    @Column (name = "volt", precision = 10, scale = 4, nullable = false)
    private BigDecimal volt;

    @NotNull
    @Column (name = "time", nullable = false)
    private double time;

    public Voltage() {
    }

    public Voltage(final BigDecimal volt, final double time) {
        this.volt = volt;
        this.time = time;
    }

    public long getId(){
        return id;
    }

    public BigDecimal getVolt(){
```

```
        this.volt = volt;
    }

    public double getTime(){
        return time;
    }

    public void setTime(final double time){
        this.time = time;
    }
}
```

Batch Configuration

Let's create a batch configuration class:

```
@Configuration
@EnableBatchProcessing
public class BatchConfiguration {
}
```

@EnableBatchProcessing enables Spring Batch features and provides a base configuration for setting up batch jobs in an @Configuration class.

We need to include two components in the above class.

```
@Autowired
```

JobBuilderFactory creates a job builder. Using *StepBuilderFactory*, Spring Batch will create a step builder and will initialize its job repository and transaction manager.

Configuring ItemReader

We will now define *ItemReader* interface for our model *Voltage* which will be used for reading data from CSV file.

```
@Bean
    public FlatFileItemReader<Voltage> reader() {
        return new FlatFileItemReaderBuilder<Voltage>()
                .name("voltItemReader")
                .resource(new ClassPathResource("Volts.csv"))
                .delimited()
                .names(new String[]{"volt", "time"})
                .lineMapper(lineMapper())
                .fieldSetMapper(new BeanWrapperFieldSetMapper<Voltage>() {{
                    setTargetType(Voltage.class);
                }})
                .build();
    }
```

Here, we are creating FlatFileItemReaderBuilder of model *Voltage.*

- *name* - Name of the *ItemReader*
- *resource* - Specify path for the resource file to be read

- *names* - Pass the fields that are to be read
- *lineMapper* - Interface to map lines from file to domain object.
- *fieldSetMapper* - Interface to map data obtained from a fieldset to an object.

Note that, we have passed custom *lineMapper()* above. Let us define that bean.

```
@Bean
    public LineMapper<Voltage> lineMapper() {

        final DefaultLineMapper<Voltage> defaultLineMapper = new DefaultLineMapper<>();
        final DelimitedLineTokenizer lineTokenizer = new DelimitedLineTokenizer();
        lineTokenizer.setDelimiter(";");
        lineTokenizer.setStrict(false);
        lineTokenizer.setNames(new String[] {"volt","time"});

        final VoltageFieldSetMapper fieldSetMapper = new VoltageFieldSetMapper();
        defaultLineMapper.setLineTokenizer(lineTokenizer);
        defaultLineMapper.setFieldSetMapper(fieldSetMapper);

        return defaultLineMapper;
    }
```

In the custom *lineMapper*, we can specify the delimiter to be read from CSV file and also used for reading string values into database-specific datatypes. The *VoltageFieldSetMapper* is defined as follows:

```
import org.springframework.batch.item.file.mapping.FieldSetMapper;
import org.springframework.batch.item.file.transform.FieldSet;
import org.springframework.stereotype.Component;

@Component
public class VoltageFieldSetMapper implements FieldSetMapper<Voltage> {

    @Override
    public Voltage mapFieldSet(FieldSet fieldSet) {
        final Voltage voltage = new Voltage();

        voltage.setVolt(fieldSet.readBigDecimal("volt"));
        voltage.setTime(fieldSet.readDouble("time"));
        return voltage;

    }
}
```

Configuring ItemProcessor

We will define the processor in Batch configuration as follows:

```
  @Bean
    public VoltageProcessor processor() {
        return new VoltageProcessor();
    }
```

We have defined a custom processor *VoltageProcessor.* Once the data is read, this processor is used for processing the data such as

```
package com.techshard.batch.configuration;

import com.techshard.batch.dao.entity.Voltage;

import org.springframework.batch.item.ItemProcessor;

import java.math.BigDecimal;

public class VoltageProcessor implements ItemProcessor<Voltage, Voltage>{

    @Override
    public Voltage process(final Voltage voltage) {
        final BigDecimal volt = voltage.getVolt();
        final double time = voltage.getTime();

        final Voltage processedVoltage = new Voltage();
        processedVoltage.setVolt(volt);
        processedVoltage.setTime(time);
        return processedVoltage;
    }
}
```

ItemWriter

Once the data is processed, the data needs to be stored in a database as per our requirement. We will define a *JdbcBatchWriter* to insert data into a database table. There is also JPA specific *JpaItemWriter* which can be used with *EntityManager*.

```
@Bean public JdbcBatchItemWriter<Voltage> writer(final DataSource dataSource) { return new JdbcBatchItemWriterBuilder<Voltage>() .itemSqlParameterSo
```

Job and Step Configuration

```
@Bean public Step step1(JdbcBatchItemWriter<Voltage> writer) { return stepBuilderFactory.get("step1") .<Voltage, Voltage> chunk(10) .reader(reader()
```

Here, *step1* is just a name of the *Step* which we can define. We can also specify chunk size in *Step* configuration.

Finally, a Job is defined as follows:

```
@Bean public Job importVoltageJob(NotificationListener listener, Step step1) { return jobBuilderFactory.get("importVoltageJob") .incrementer(new Run
```

Note that we have passed *NotificationListener* that extends Spring Batch's *JobExecutionListenerSupport*. It can log results before or after job execution. Here, we have only defined *afterJob()*. *JobExecutionListenerSupport* also provides *beforeJob()* to log any information before the job execution.

```
package com.techshard.batch.configuration; import com.techshard.batch.dao.entity.Voltage; import org.slf4j.Logger; import org.slf4j.LoggerFactory; i
```

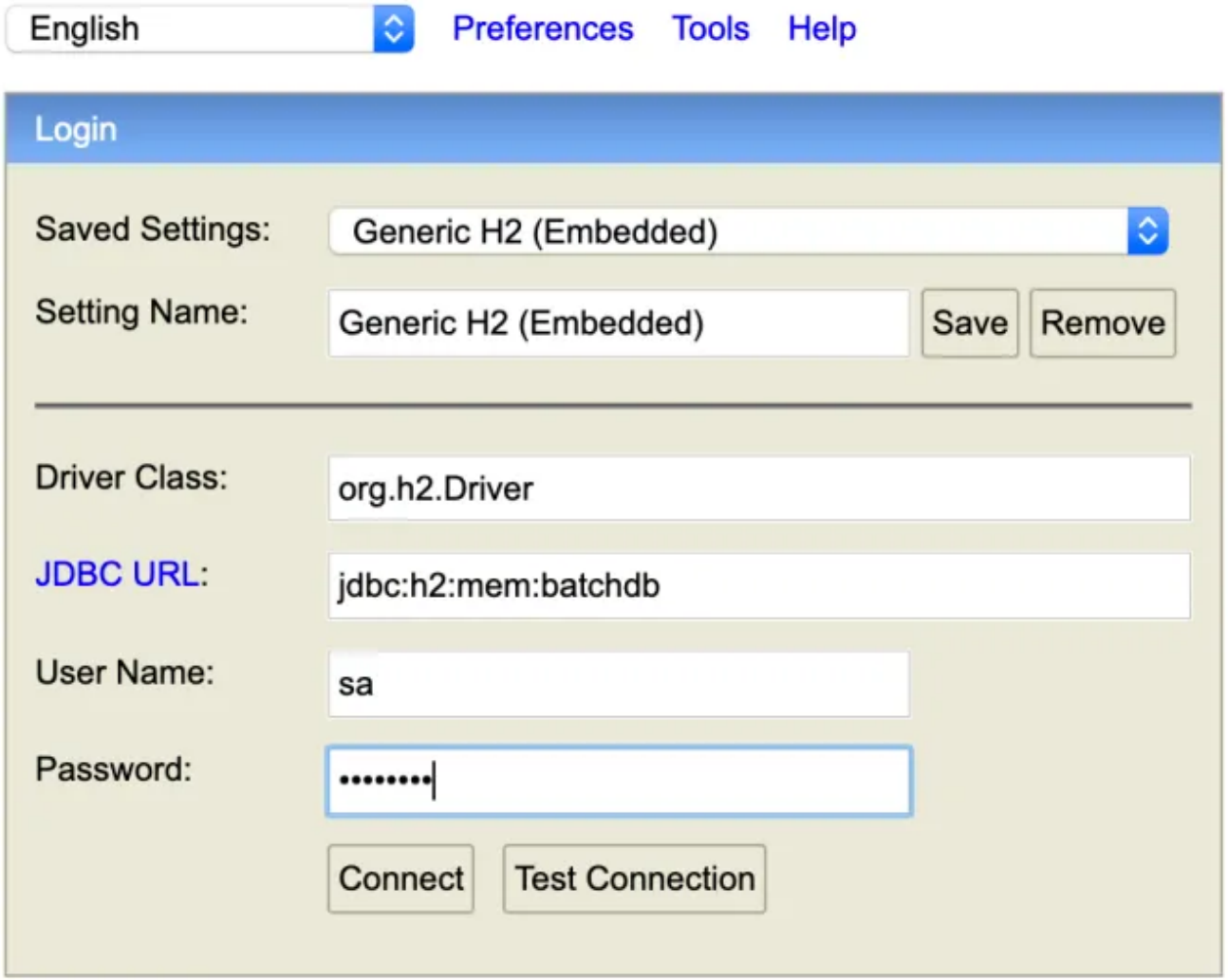
Before we run the application, we will enable H2 (in-memory) console in application.properties.

```
spring.datasource.url=jdbc:h2:mem:batchdb spring.datasource.driverClassName=org.h2.Driver spring.datasource.username=sa spring.datasource.password=p
```

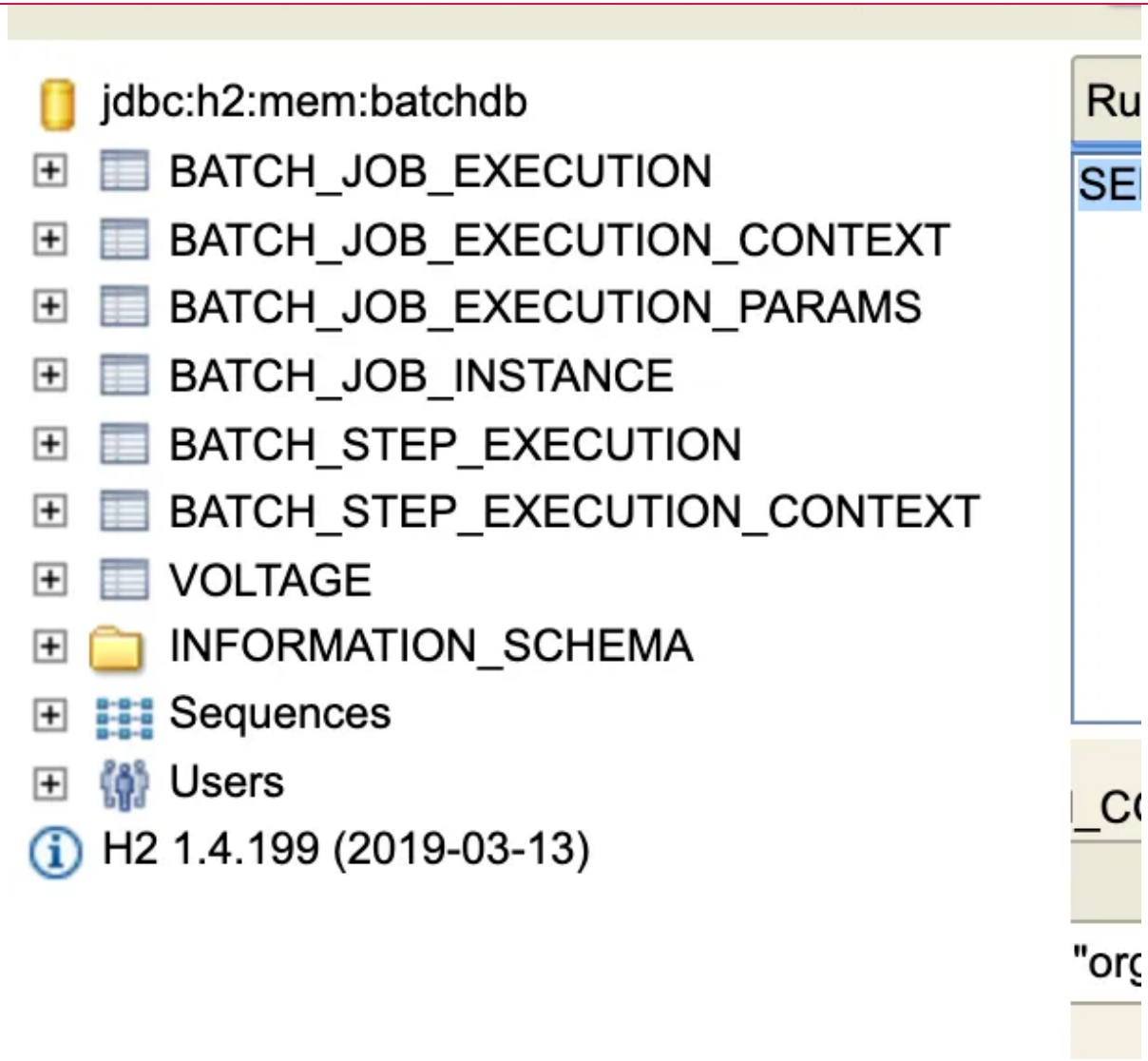
Additionally, I have also configured Aspect using Spring AOP to measure the time taken by batch execution.

```
package com.techshard.batch; import org.aspectj.lang.ProceedingJoinPoint; import org.aspectj.lang.annotation.Around; import org.aspectj.lang.annotat
```

Running the Application



Once we login, we will be able to see the table *Voltage* and all the tables created by Spring Batch. In these tables, we will find all the details about job execution such as job name, status, id and so on.



Conclusion

The complete code can be found on my GitHub repository.

Like This Article? Read More From DZone

DZone Article

Batch Processing With Spring Batch and AMQP: Easier Than You Think

DZone Article

Spring Batch CSV Processing

DZone Article

How to Prevent OutOfMemoryError When You Use @Async

Free DZone Refcard

Visual Testing

Topics: BATCH PROCESSING , PERFORMANCE , SPRING BATCH , SPRING BOOT , TUTORIAL