# Spring batch

Processing finite amount of data/ batch of data without interaction or interruption

We can restart if that chunk fails

Some important points

This is not like a Kafka application which will run continuously,

This will start and it will do assigned work and it will die

In real time, through autosys tool application will be started/launched freshly and it will complete its work in 15-120 minutes approx. and it will die

### **When to use spring batch**

If you are handling with huge volume of data then use spring batch – because it can process chunk by chunk

1. Suppose if bank wants to send BE-ALERT messages or current balance mails like our Citibank then use spring batch , because spring batch can fetch person records chunk by chunk and send the messages to the chunk of users-- else u cant send msgs to all people at once, if u do jvm will blast.
2. All the transactions happening under RTGS, NEFT , all the transactions will be stored and executed as a batch.
3. Some vulnerability scanner which will scan bulk files

Use cases:- generation of email statements, weather forecasting reports

Ex:- CSV to database

Item Reader

Item Processor

Item Writer

Step

Job

Job launcher

Job Repository

All the job status will be stored in job repository job ran or failed or the status ..& when job started & when job ended , job repo will store all that data into db

Features of spring batch

1. It will provides us the start job, stop the job options
2. Retry and skip mechanism

### Step object

1 job will have lot of steps, it can have either 1 or 100 steps, job is the main task , main task will have all

Here main task is money transfer, but it will have all these below steps

Create step using stepBuilderFactory (This factory will be available directly we can autowire directly)

1 single chunk oriented step contains 3 task 1 step= ItemReader+ItemProcessor+ItemWriter

@Autowired  
StepBuilderFactory **stepBuilderFactory**;  
return **stepBuilderFactory**.get("S1")  
 .<StudentJsonPOJO,StudentJsonPOJO>chunk(4)  
 .reader(itemReader(null))  
 .writer(*itemWriter*)  
 .build();

Debiting from 1 ac

Crediting to target account

Sending sms as transaction status

#### StepExecutionContext

1. It is the memory that is available across the entire step, for ex:- (components of step are – ItemReader, ItemProcessor, ItemWriter…)

U can set from ItemReader and use in ItemWriter

Here all the data which we set to Stepexecution\_context– will be stored in database once that step is completed immediately

For each step there will be separate step execution context (context is like memory).

And each step execution context data stored in table called stepexecution\_context

Because each step can have ItemReader, ItemProcessor, ItemWriter with this we can set values in ItemReader and we can fetch in ItemWriter

This is the memory available for that entire step.

This is also a map, which will be stored in database table

BATCH\_STEP\_EXECUTION\_CONTEXT refer schema-oracle10.sql in spring-batch-core-\*.jar all these sql files are present in that jar, dropping the

StepExecutionListener

This will be executed before and after executing the step

1. Configure step exec listener to a step 🡪Just creating stepExecutionListener is not enough**, we need to link it to a step,** because in 1 proj we will have multiple steps, and multiple listeners,

We have to configure that listener to a particular step, like which listener should fire for which step execution, so we need to configure

1. All the data set to stepexecution\_context will be stored in table “SELECT \* FROM sprbatch.batch\_step\_execution\_context;

Dividing into chunks is mandatory, else memory out of bounds exceptions only, it says how many records u want to process at a time

|  |  |
| --- | --- |
| */\*\*  \* creating step exec listener is not enough we only need to configure/map this listener to a step  \*/* @Component public class MyStepExecListener implements *StepExecutionListener* {  public static String *companyName* = "companyName";  @Override  public void beforeStep(StepExecution *stepExecution*) {  System.***out***.println("firing step execution listener & method name is - beforeStep()");  ExecutionContext stepExecContext = *stepExecution*.getExecutionContext();  stepExecContext.put(*companyName*,"Tata consultancy Services");  }   @Override  public ExitStatus afterStep(StepExecution *stepExecution*) {  System.***out***.println("firing step execution listener & method name is - afterStep()");   return ExitStatus.***COMPLETED***;  } } | @Autowired MyStepExecListener **myStepExecListener**;  @Bean public *Step* taskletOrientedSecondStep(){  return **stepBuilderFactory**.get("Second step idi telusa")  .tasklet((*contribution*,*chunkContext*)->{  *Map*<String, Object> jobExecutionContext = *chunkContext*.getStepContext().getJobExecutionContext();  System.***out***.println("Executing the step 2");  System.***out***.println("fetching value from job context -->"+ jobExecutionContext.get(MyJobListener.*appName*));  ExecutionContext stepExecutionContext = *chunkContext*.getStepContext().getStepExecution().getExecutionContext();   System.***out***.println("fetching value from step context --> "+ stepExecutionContext.get(MyStepExecListener.*companyName*));  return *RepeatStatus*.***FINISHED***;  })  *//here we configured our listener to a particular step* .listener(**myStepExecListener**)  .build(); } |

Tasklet step

Mostly we wont use tasklet step, we will use chunk oriented step

Like if u have some job just to call a micro service, so to print 1-100 element, for that as we don’t need itemreader, ItemWriter then

In that cases instead of Chunk oriented step we can use Tasklet

*//Creating a tasklet step and use that step in the job* @Bean  
 public *Step* taskletOrientedStep(){  
 return **stepBuilderFactory**.get("Tasklet oriented step")  
 .tasklet(new *Tasklet*() {  
 @Override  
 public *RepeatStatus* execute(StepContribution *contribution*, ChunkContext *chunkContext*) throws Exception {  
 System.***out***.println("My Tasklet oriented step");  
 return *RepeatStatus*.***FINISHED***;  
 }  
 }).build();  
 }  
   
*// use the above step for our job* @Bean  
 public *Job* taskletJob(*Step taskletOrientedStep*){  
 System.***out***.println("Building a tasklet oriented stepped job");  
 return **jobBuilderFactory**.get("Tasklet oriented job")  
 .start(*taskletOrientedStep*)  
 .next(*taskletOrientedStep*) //wantedly used same task in next step  
 .build();  
 }

#### Chunk oriented step

ItemWriter will insert chunk of records at once into database, if it inserts insert one by one right then performance will be impacted

If Chunk size =4

1. 4 items will be read continuously
2. 4 items will be processed continuously at a time
3. 4 items will be inserted into database at once

ItemReader will read the chunk (small size) of records from the excel file- means it will read until chunk size is met from file- here read method will be invoked until chunk size is reached.so that’s why configure chunk size for each and every step, If u don’t configure itemreader will keep on reading and it will finally throw exception

ItemProcessor will be called for each and every item present in the chunk

### Job

It is the main task ex:- sending data from one to another destination, 1 job will have many steps

Think climbing steps Is a job- so that job will have multiple steps, not a single step.

Job is nothing but a process which run from start to finishes without any interruption

Job is also created by jobBuilderFactory.

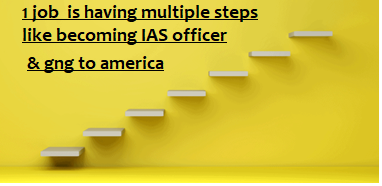
Job is broken up into steps- each step is independent,

Job means lot of steps – like if u want to become IAS officer u have lot of steps

* We can have a job with 🡪 Tasklet and chunk oriented step combo

Using job builder factory we will create job (jobBuilderFactory bean is readily created by spr batch auto configuration, we just need to autowire it)

@Autowired  
JobBuilderFactory **jobBuilderFactory**;

   
*// use the above step for our job, Here to create 1 job it involves lot of steps* @Bean  
 public *Job* taskletJob(*Step taskletOrientedStep*){  
 System.***out***.println("Building a tasklet oriented stepped job");  
 return **jobBuilderFactory**.get("Tasklet oriented job")  
 .start(*taskletOrientedStep*) //create Step-1  
 .next(*taskletOrientedStep*) //create Step -2  
 .build();  
 }

Ex:- to get software job

1. we have to complete B.Tech
2. learn all trainings in ameerpet
3. prepare resume
4. attend interviews

#### JobExecutionContext

1. It is the memory that is available across all the steps, for ex:- if u want to use step-1 data in step-2 , then set the data to context and fetch in step2 from step execution context
2. Remember all that data which we set to JobExecutionContext will be saved in db by spring batch framework in table “batch\_job\_execution\_context”



Using below, if u set something to jobContext

ExecutionContext executionContext = *jobExecution*.getExecutionContext();  
 executionContext.put(*appName*,"CTRPC");//setting values to context

Then this is available as per above, means from chunk context we can get stepcontext from which u can get jobexecutioncontext

public *Step* taskletOrientedStep(){  
 return **stepBuilderFactory**.get("idi 1st step ayy swami telusa")  
 .tasklet(new *Tasklet*() {  
 @Override  
 public *RepeatStatus* execute(StepContribution *contribution*, ChunkContext *chunkContext*) throws Exception {  
 System.***out***.println("Executing My Tasklet oriented step");  
 ***Map*<String, Object> jobExecutionContext = *chunkContext*.getStepContext().getJobExecutionContext();** return *RepeatStatus*.***FINISHED***;  
 }  
 }).build();  
}

#### JobExecutionListener

Note: - just creating listener as a spring bean will not be enough, we should link that listener to our job,

Because in our app, we can have 2-10 jobs also, we can have 3 to 4 listeners, so if we just declare a listener, it cannot be listener for every job right,

**So we should link a listener to that job**

This must be a spring bean, so better mark it as a spring bean, marking is not enough

We need to attach to the job

|  |  |
| --- | --- |
| @Autowired MyJobListener **myJobListener**;  @Bean public *Job* taskletJob(*Step taskletOrientedStep*,*Step taskletOrientedSecondStep*){  System.***out***.println("Building a tasklet oriented stepped job");  return **jobBuilderFactory**.get("Tasklet oriented job2")  .incrementer(new RunIdIncrementer())  .listener(**myJobListener**) //configure our listener to job  .start(*taskletOrientedStep*)  .next(*taskletOrientedSecondStep*)  .build(); } | @Component public class MyJobListener implements *JobExecutionListener* {   public static String *appName* = "app\_name";  @Override  public void beforeJob(JobExecution *jobExecution*) {  ExecutionContext executionContext = *jobExecution*.getExecutionContext();  executionContext.put(*appName*,"CTRPC");//setting values to context  System.***out***.println("firing job execution listener & method name is - beforeJob()");  }   @Override  public void afterJob(JobExecution *jobExecution*) {  System.***out***.println("firing job execution listener & method name - afterJob()");  System.***out***.println("fetching value from jobExecContext with key app\_name, value is--> "+*jobExecution*.getExecutionContext().get("app\_name"));  } } |

Job Instance & Job Execution &

Job instance – means 1 day only 1 job instance – , for (1 instance, 1 execution) 1 job instance there will be only 1 successful execution

If a job is instantiated for that day

if that job execution failed, then it can re-start that job , then there will be another job execution

Job

Job Instance 2-May

Job Instance 1-May

Job Execution 1-May - restarted

Job Execution 1-May failed

Under 1 instance, there will be only 1 successful execution, 2nd time job wont execute as previous execution is success

Job param builder

1. Motto is , we cannot run the same job with same parameter, everytime we should launch a job with different parameter

Use case:- if u want to run job daily only once, then take current date as job parameter, so that every day the job will run with different parameters

All the job parameters or step parameters which we set will always be saved in database

1. We have to tell which job we have to launch with which parameters
2. Either u get data from environmental variables or from main method its always better to set those data to job parameters so that

Those parameters will be stored in table named “batch\_job\_execution\_params”

1. JobParameters parameters=new JobParametersBuilder() , this JobParameters is just a hashmap  
   we have to link those JobParameters with a certain job like this 🡪 **launcher**.run(**getJobA**,parameters);

|  |  |
| --- | --- |
| But, if u want to run that job everytime u start then  1) Either We have to run the **same job with diff parameters or**  **2) we have to use incrementer**  **jobBuilderFactory**.get("getJobA")  .incrementer(new RunIdIncrementer())  This will make “diff job instance id”  Means even if u run the same job 2nd time for that day,  That same job will have a diff job instance    Here if u see job key will be diff , so even if u run the job 2nd time on a day it will run perfectly, now here it will assume like its running on another day | @Component public class MyItemRunner implements *CommandLineRunner* {  @Autowired  *JobLauncher* **launcher**;   @Autowired  *Job* **getJobA**;   @Override  public void run(String... *args*) throws Exception {  JobParameters parameters=new JobParametersBuilder()  .addLong("time", Instant.*now*().toEpochMilli()).toJobParameters();  **launcher**.run(**getJobA**,parameters);   } }  //Above means we are running that specific job with diff parameters,  each time the job run arg will be diff, hence it will run  So as per above code , **we cant run same job with same parameters if u run u will get exceptions, saying job already run with this parameters**  *A job instance already exists and is complete for parameters={time=1}. If you want to run this job again, change the parameters.*  (not sure if it is forever or not)  We can run same job with diff parameters   1. (Those parameters will be stored in a table called **batch\_job\_execution\_params and correspondingly entries will be created in jobinstance and job execution tables**)   Note:- generally we can run a job with multiple parameters  Not only this, if u change the step name then also it will run on same day second time |
| Using Incrementer  return **jobBuilderFactory**.get("getJobA")  .incrementer(new RunIdIncrementer())  This incrementer is to provide diff values on running each and every time  if we configure this incrementer, we have a LONG type key called “run.id” initiallly value will be 1 & all these job parameters will be stored in a table called “batch\_job\_execution\_params”  2nd time the value for key named “run.id” is set to 2 by incrementing  even if u run with same parameters, or even If u don’t send any parameters also fine, bec it will increment the value for “run.id” | See here these job parameters which we have set are stored in |
|  |  |

Fethcing job parameters

Way-1 using chunk context

public RepeatStatus execute( StepContribution step, ChunkContext context ) throws Exception {

String name = (String) **context.getStepContext().getJobParameters()**.get("name");

System.out.println( String.format(HELLO\_WORLD, name) );

return RepeatStatus.FINISHED;

}

Way-2 annotation way

If u are fetching using #{} symbol, first u have to set to job parameters before u launch the job

@Override  
public void run(String... *args*) throws Exception {  
 System.***out***.println("came to job launcher class");  
 JobParameters jobParameters = new JobParametersBuilder()  
 .addString(*location*, **environment**.getProperty("fileLocation"))  
 .addLong("run.id", LocalTime.*now*().toNanoOfDay())  
 .toJobParameters();  
 System.***out***.println("location of that file is " + **environment**.getProperty(*location*));

//Launch that particular job with those job parameters  
 **jobLauncher**.run(**job**, jobParameters);  
}

Fetching those parameters----------

@StepScope  
 @Bean  
 public FlatFileItemReader<Student> flatFileItemReader(  
 @Value("#{jobParameters['location']}") String *location*) {  
 System.***out***.println("creating bean of item Reader with this location"+ *location*);  
 FileSystemResource fileSystemResource=null;  
*// fileSystemResource=new FileSystemResource("E:\\study related\\my git hub -new\\Practice-projects\\spring\\batch\\4.ItemReaders Demo\\inputFiles\\students.csv");* fileSystemResource=new FileSystemResource(*location*);  
*// System.out.println("key value is "+pathToFile);* System.***out***.println("file name is -->"+fileSystemResource.getFilename());

Way -3 –XML way

<beans:bean id="inputFile"

class="org.springframework.core.io.FileSystemResource" scope="step">

<beans:constructor-arg value="#{jobParameters[inputFile]}"/>

</beans:bean>

Job launcher

This is to start the actual job by taking some inputs from job parameters

public class JobLauncherWithJobParams implements *CommandLineRunner* {  
 @Autowired  
 *JobLauncher* **jobLauncher**;  
 @Autowired  
 *Job* **job**;  
 @Autowired  
 *Environment* **environment**;  
 String **location** = "location";  
  
 @Override  
 public void run(String... *args*) throws Exception {  
 System.***out***.println("came to job launcher class");  
 JobParameters jobParameters = new JobParametersBuilder()  
 .addString(**location**, **environment**.getProperty(**location**))  
 .addLong("run.id", LocalTime.*now*().toNanoOfDay())  
 .toJobParameters();  
 System.***out***.println("location of that file is " + **environment**.getProperty(**location**));  
 **jobLauncher**.run(**job**, jobParameters);  
 }  
}

Job Repository

All the job statuses is maintained in the job repository, so that if any chunk fails , the job can restart from there

When job started, when it’s ended, status –success or failed

### Item Reader or Item Processor vs Item Writer

Item is nothing but an individual record.

If 25 times reader executed, then 25 times Processor will be executed, whereas only 3 times the writer will be executed. Because 25 times the writer should not hit the database, we will mention the chunk size as 10, only when it has 10 records then it will hit the ItemWriter.

### ItemReader

In most of the scenarios we will use predefined readers only

ItemReader –the powerful feature is it will read chunk of data continuously

ex:- FlatFileItemReader (predefined ItemReader given by batch api used to read data from file and convert to objects)

Ex:- If chunk size is 20 , that item reader read() method will be executed 20 times continuously / will be reading until chunk size is met/ until chunk of records are read,

And it will keep on running/executing the reader until last element is null.

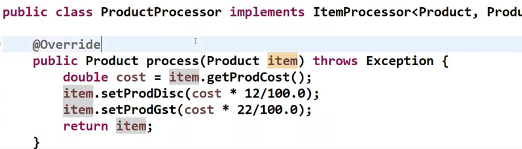
And of course, for each record it will hit ItemProcessor, so 20 times it will hit ItemProcessor

Ex:- reading data from csv and writing to MySQL

### ItemProcessor

The item processor is going to hold the output until the chunk size is met.

If there is no business logic, u can have a chunk oriented step **without** item processor.



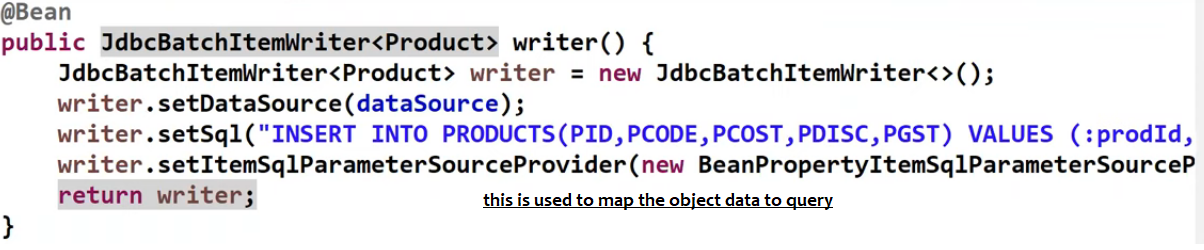
### ItemWriter

Here use predefined writer, it will receive the chunk of data at once, & it will write that full chunk at once to database

Ex:- JDBCBatchItemWriter – this is the predefined class given by batch to write data to database

ItemWriter will be invoked only once for that chunk –if chunk size is 10, it will be invoked once for that chunk

For that 10 records, because if item writer is writing to that db, it should not write each record once, it should write only once for that chunk



### Tables

|  |  |
| --- | --- |
| batch\_job\_execution\_params | Here we can see all the parameters passed to run that particular job |
| batch\_job\_execution | Summary or status of job execution |
| batch\_step\_execution | Step wise execution status |
| batch\_job\_instance | Job name and job key |

### Strategy

To process millions of records, we don’t process them all at a time,

We process them in chunks and stores in database

### Coding

Jars

<dependency>  
 <groupId>com.mysql</groupId>  
 <artifactId>mysql-connector-j</artifactId>  
 <scope>runtime</scope>  
</dependency>

To work with my sql we should add above jar

#### App properties

spring.batch.jdbc.initialize-schema =always

means everytime it wont create tables – if tables are already present it will leave as it is, but is absent then it will create tables required for spring batch

Annotations

1. @EnableBatchProcessing

## ItemReaders

### FlatFileItemReader

Line mapper to map the lines and 1st heading column,

U have to **set line Tokenizer** and **field set mapper**

Item Reader

Line Mapper

In my practice, only hard coded path is working, unable to get via @Value using job parameters

Best way: - never send the path through environment variable, instead if we set to job parameters then those arguments will be saved in job parameters table also

We can see the arguments passed, so remember to set the values to job parameters even if we receive arguments from main method

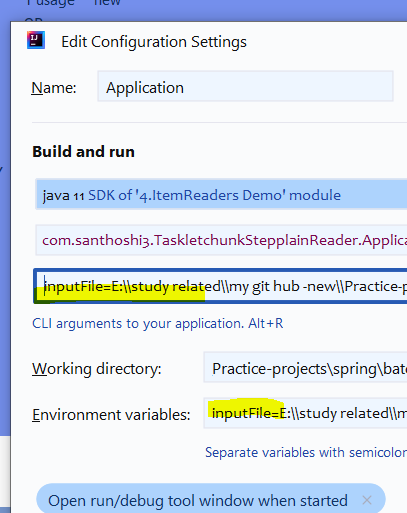
Think always in prod point of view, in prod we have autosys, we will run job through autosys, so even if we directly get it from environment variables we have

To set to job parameters

@StepScope  
 @Bean  
 public FlatFileItemReader<Student> flatFileItemReader(  
 @Value("#{jobParameters['inputFile']}") FileSystemResource *fileSystemResource*) {  
*// public FlatFileItemReader<Student> flatFileItemReader( @Value("#{jobParameters['inputFile']}") String pathToFile) {  
 fileSystemResource*=new FileSystemResource("E:\\study related\\my git hub -new\\Practice-projects\\spring\\batch\\4.ItemReaders Demo\\inputFiles\\students.csv");  
*// System.out.println("key value is "+env.getProperty(location));  
// fileSystemResource=new FileSystemResource(env.getProperty(location));  
// System.out.println("key value is "+pathToFile);* System.***out***.println("file name is -->"+*fileSystemResource*.getFilename());  
  
 System.***out***.println("creating a item reader object as bean");  
 FlatFileItemReader<Student> itemReader=new FlatFileItemReader<>();  
 itemReader.setLinesToSkip(1);  
 itemReader.setResource(*fileSystemResource*);  
 *AbstractLineTokenizer* tokenizer =new DelimitedLineTokenizer();  
 tokenizer.setNames("ID","First Name","Last Name","Email");  
  
 BeanWrapperFieldSetMapper<Student> fieldSetMapper= new BeanWrapperFieldSetMapper<Student>();  
 fieldSetMapper.setTargetType(Student.class);  
  
 DefaultLineMapper lineMapper= new DefaultLineMapper();  
 lineMapper.setLineTokenizer(tokenizer);  
 lineMapper.setFieldSetMapper(fieldSetMapper);  
  
 itemReader.setLineMapper(lineMapper);  
 System.***out***.println("Item reader bean is created successfully");  
 return itemReader;  
  
 }

JsonItemReader

@Value("${inputFile}") // we have given the from intellij run arguments those arguments are set to spring context  
 String **fileLocation**;  
  
 @Bean  
 @StepScope  
 public JsonItemReader<StudentJsonPOJO> itemReader(  
 @Value("#{jobParameters['inputFile']}") FileSystemResource *fileSystemResource*) {  
 System.***out***.println("parameters are -->" +*fileSystemResource*);  
 System.***out***.println("file location passed to vm args is --> "+**fileLocation**);  
*// fileSystemResource=new FileSystemResource("E:\\study related\\my git hub -new\\Practice-projects\\spring\\batch\\4.ItemReaders Demo\\inputFiles\\students.json");  
 fileSystemResource*=new FileSystemResource(**fileLocation**);  
  
 System.***out***.println("creating itemReader bean");  
 JsonItemReader<StudentJsonPOJO> itemReader = new JsonItemReader<>();  
 itemReader.setResource(*fileSystemResource*);  
 itemReader.setMaxItemCount(10);  
 itemReader.setCurrentItemCount(2);  
 itemReader.setJsonObjectReader(new JacksonJsonObjectReader<>(StudentJsonPOJO.class));  
 return itemReader;  
 }



we should always give path with with double back slash

DatabaseItemReader

@Bean  
public JdbcCursorItemReader<StudentDbPOJO> jdbcCursorItemReader(){  
 System.***out***.println("creating JDBCitemReaderBean to read from database");  
 JdbcCursorItemReader<StudentDbPOJO> itemReader=new JdbcCursorItemReader<>();  
 itemReader.setDataSource(**dataSource**);  
 itemReader.setSql("select id, first\_name as firstName , last\_name as lastName, email from student");  
 BeanPropertyRowMapper<StudentDbPOJO> rowMapper = new BeanPropertyRowMapper<>();  
 rowMapper.setMappedClass(StudentDbPOJO.class);

itemReader.setCurrentItemCount(5); //to skip first 4 items

itemReader.setRowMapper(rowMapper);  
 return itemReader;  
}

Here we are giving query and DataSource, and mentioning the chunk size as 3, to fetch 3 records we have to understand how many times it is hitting database

In 1 hit is it fetching 3 records? Or in 1 hit is it fetching only 1 record?

Connect with multiple databases

1. One database must be primary
2. Create DataSource using DatasourceBuilder.create().build()
3. Add new properties in properties file

|  |  |
| --- | --- |
| Create properties for 2 databases with  spring.old-datasource.jdbc-url=jdbc:mysql://localhost:3306/sprbatch spring.old-datasource.username=mani spring.old-datasource.password=mani spring.old-datasource.driverClassName=com.mysql.cj.jdbc.Driver  spring.new-datasource.jdbc-url=jdbc:mysql://localhost:3306/aims spring.new-datasource.username=santu1 spring.new-datasource.password=santu1 | And bind all those properties to Datasource beans  @Bean("primaryDataSource") @Primary @ConfigurationProperties("spring.old-datasource") public *DataSource* createMainDataSource(){  System.***out***.println("creating data source");  *DataSource* build = DataSourceBuilder.*create*().build();  return build; } @Bean(value = "secondaryDataSource") @ConfigurationProperties("spring.new-datasource") public *DataSource* secondaryDataSource(){  System.***out***.println("creating secondary data source");  *DataSource* build = DataSourceBuilder.*create*().build();  return build; } |

And attach whichever DataSource u want to ItemReader

@Bean  
public JdbcCursorItemReader<StudentDbPOJO> jdbcCursorItemReader(@Autowired *DataSource dataSource*){  
 System.***out***.println("creating JDBCitemReaderBean to read from database" +*dataSource*.toString());  
 JdbcCursorItemReader<StudentDbPOJO> itemReader=new JdbcCursorItemReader<>();  
 itemReader.setDataSource(*dataSource*);