Job

===

=>Job represents work to be completed.. Each Job contains 1 or more steps (Minimum 1 step will be there) =>Step object means it is the object of java class that implements <pkg>.Step(1). org.springframework.batch.core.Step) =>Job object means it is the object of java class that implenetns pkg.Job(1) (org.springframework.batch.core.Job)

=> Job represents total work to be completed where as step represents task/sub task of the Job

in month

eg: Job (Finding the Customers who are having due date this to renew polocies and triggering the SMS messages)

=>step1 (sub task1)

=> Get all customer having due date from

oracle DB s/w to Excel sheet

=>step2) Convert Excel sheet data to mysql DB table

Λ

OF

2005

Enduser

Of

Enduser

(becoz SMS Trigger app expeceting details in Mysql Db table)

=>step3) Trigger SMS messages..

(Read from mysql Db and trigger SMS messages)

execute()

Job

Step 1

Step 2

Step 3

execute()

KH

ExitStatus

K+

execute()

K

ExitStatus

execute()

=>One Job can contain 1 or more steps Job like task

Steps are like sub tasks

The steps will execute sequentially one after another

Job with steps

ExitStatus

ExitStatus

sequence flow diagram

Spring Batch Launch Environment

Batch Application Style - Interactions and Services

Scheduler Config.

`Job Script`

JobRunner JobLauncher

Request

JobLocator

Custom Application Artifacts

Application Architecture Services

Applications, App Servers, VMs

Job Tier

Job

Configuration

App Comant

Job Repository

Application Tier

Data Tier

ItemReader

Read

Data Access

Business

`Logic`

(ItemProcessor)

Config

Write

BemWriter

The application style is organized into four logical tiers, which include Run, Job, Application, and Data tiers. The primary goal for organizing an application according to the tiers is to embed what is known as "separation of concerns" within the system. Effective separation of concerns results in reducing the impact of change to the system.

Run Tier: The Run Tier is concerned with the scheduling and launching of

the application. A vendor product is typically used in this tier to allow time-based and interdependent scheduling of batch jobs as well as providing parallel processing capabilities.

Job Tier: The Job Tier is responsible for the overall execution of a batch job. It sequentially executes batch steps, ensuring that all steps are in the correct state and all appropriate policies are enforced.

Application Tier: The Application Tier contains components required to execute the program. It contains specific modules that address the required batch functionality and enforces policies around a module execution (e.g., commit intervals, capture of statístics, etc.)

(batch size)

Data Tier: The Data Tier provides the integration with the physical data sources that

might include databases, files, or queues. Note: In some cases the Job tier

can be completely missing and in other cases one job script can start several batch job instances.

get

to

Once we add "spring-boot-starter-batch" to the spring boot projects we multiple objects related spring batch processing through Autoconfiguraiton like

This is upto

spring boot 2.x

a)StepBuildFactory (required to create Step object) b)JobBuilderFactory (required to create Job object) deprecand moved c)JobLauncher (requred to run the Job)

but same are

spring boot 3.x

ItemReader

==========

and etc..

=> These 3 objects can be injected to our choice spring beans

to get required objects and to perform batch processing activities

In Spring boot 3.x of batch processing we get only these objs through auto configurations

===

==========

a) JobRepository obj b) JobLauncher obj c) Transaction Manager obj

====

To create Job, Step Objects, we need to use two param constructor of JobBuilder class and StepBuilder class manually org.springframework.batch.item.ItemReader<T> (1)

=> All Item Reader classes in spring batch are the implementation classes of

=> We generally do not implement ItemReader(1) i.e we do not need to develop Custom ItemReaders .. becoz spring batch has provided multiple pre-defined ItemReaders

=>The read() of Each ItemReader reads info from source repository (like file, DB and etc...) and gives either String object or Model class object representing each record of the Info.

=>The readymade ItemReaders are

=========

AvroItemReader, FlatFileItemReader, HibernateCursorItemReader, HibernatePagingItemReader, ItemReaderAdapter, ItemReaderAdapter, IteratorItemReader, JdbcCursorItemReader, JdbcPagingItemReader,

JmsItemReader, JpaCursorItemReader, JpaPagingItemReader, JsonItemReader, KafkaItem Reader, LdifReader, ListItemReader, MappingLdifReader, MongoItemReader, MultiResourceItemReader, Neo4jItemReader, RepositoryItemReader, ResourcesItemReader, SingleItemPeekableItemReader, StaxEventItemReader, StoredProcedureItemReader, Synchronized ItemStreamReader and etc..

=>Since all ItemReaders are pre-defined classes we cfg them as spring bean using

@Bean methods of @Configuration class /Main class (which internally @Configuration class)

In AppConfig.java (Configuraiton class) (Annotated with @Configuration)

==========

package com.nt.config:

@Bean(name="fflReader")

public ItemReader createReader(){

FlatFileItemReader reader=new FlatFileItemReader<...>();

---

return reader; }

ItemWriter

==========

=>It is given to write given chunk/batch of information to Destination ..

=> Generally we do not develop ItemWriters.. becoz there are multiple readyMade ItemWriters to use

=>All ItemWriters are impl classes of org.springframework.batch.itemItemWriter<T>. (1)

The ready made ItemWriters are

=========

==========

AmqpItemWriter, AsyncItemWriter, AvroItemWriter, ChunkMessageChannelItemWriter, ClassifierCompositeItemWriter, CompositeItemWriter, FlatFileItemWriter, GemfireItemWriter, HibernateItemWriter, ItemWriterAdapter, ItemWriterAdapter, JdbcBatchItemWriter, JmsItemWriter, JpaItemWriter, JsonFileItemWriter, KafkaItemWriter, KeyValueItemWriter, ListItemWriter, MimeMessageItem Writer, MongoItemWriter, MultiResourceItemWriter, Neo4jItemWriter, PropertyExtractingDelegatingItemWriter, RepositoryItemWriter, SimpleMailMessageItemWriter, SpELMappingGemfireItem Writer, StaxEventItemWriter, SynchronizedItemStreamWriter

=>since all the ItemWrites are pre-defined classes, we generally configure them using @Bean methods of @Configuration class or main class..

In AppConfig.java (Configuration class)

@Bean(name="jbiwriter")

public ItemWriter createWriter(){

JdbcBatchItemWriter<Customer> writer=new JdbcBatchItemWriter();

usecase::

Convert CSV file (Flat) data to ing

DB table records after filter the records based on bill amount.

return writer; }

ItemProcessor

==========

CSV file (Comma Seperated Value File) == Ms Excel file

=> It is impl class of org.springframework.batch.item.ItemProcessor<1,0> (1)

=> Very Limited ready made ItemProcessors are avaiable.. So we generally develop custom Item Processors.

ready made ItemProcessor are

==========

AsyncItemProcessor, BeanValidatingItemProcessor, ClassifierComposite ItemProcessor, CompositeItemProcessor, FunctionItemProcessor, ItemProcessorAdapter, ItemProcessorAdapter, PassThroughItemProcessor, ScriptItemProcessor, ValidatingItemProcessor

with

=>The <I> ofItemProcessor must match <T> of ItemReader and similary the <0> of ItemProcessor must match with <T> of ItemWriter

CustomerFilterItemProcessor.java

@Component ("processor")

public class CustomerFilterItemProcessor implements ItemProcessor<String,Customer>{

Model class name

public Customer process(String info){

(0)

(1)

}

//logic for convertion and fltering

(String to customer obj)

return Customer object;

to give to give the processed data as Model class objs to Destination (Custmer class obj)

(output type)

form CSV file /Excel file

(input type)

csv file/excel

file read each

DB s/w

writes of

List of Customer objs

Batch

(JdbcCursorItemWriter)

Step

=> It is impl class object of org.springframework.batch.core.Step (1)

=> Each Step object represents one task/sub task of a Job

record as String App

(FlatFileItemReader

CustomerItemProcessor process each

record from CSV file(String) and creates

Customer obj(Model class obj)

after filtering the records based on the bill amount

=> Every Step object must be linked with 1 reader object,1 writer object and 1 processor object.

=> We generally create Step object by Using the StepBuilderFactory object that comes spring boot App throrugh AutoConfiguration.

=>every Step object must contain the following 5 details

a)step name (logical name)

b) <input type, output type> + chunksize (batch size)

c) reader obj

d) writer obj

(builder Design Pattern)

Builder DesignPattern says create complex object by using bunch of small objects by defining certain process.. So that process can be used to create different objs of the same category

e) processor obj

=>We generally use @Bean method in @Configuration class to create Step object with StepBuilderFactory object and poviding

the above 5 details..

In AppConfig.java (Confiuration class) (In spring boot 2.x)

==========

@Configuration

==========

@ComponentScan(basePackages=".....")

public class AppConfig{

@Autowired

private StepBuilderFactory sbFactory;

@Autowired

private JobBuilderFactory jbFactory; @Autowired

came through

AutoGonfiguration

private CustomerFilterItemProcessor custProcessor;

user-defined

spring bean

method chaining

==========

calling method().method().method()... is called Method Chaining i.e The returned object of

1 method will be used as base object to call another method.

}

@Bean(name="fflReader")

public ItemReader createReader(){

reader

FlatFileItemReader reader=new FlatFileItemReader<...>();

. . . . .

return reader;

}

@Bean(name="step1"),

public Step createStep() {

writer

@Bean(name="jbiwriter")

public ItemWriter createWriter(){

JdbcBatchItemWriter<Customer> writer=new JdbcBatchItemWriter();

---

return writer;

}

return sbFactory.get("step1") //step name

.<String,Customer>chunk(10) //chunk size + Input, output types .reader(createReader()) //reader obj

method chaining .processor(custProcessor) //processor obj

and builder dp

is used

}

.writer(createWriter()) //writer obj

.build(); // build() method builds the Step obj

....

// job configuration

....

reader obj

Logical Name

writer obj

Step object

processor obj

chunk size

<1,0>

JobExecutionListener (1)

==========

=>By implementing this interface, we can perform event handling on job activities like when

is started, when job is completed, what is status of job completion (Success or failure) and etc..

Job

the

=> In creation fob object we need JobExecutionListener obj

org.springframework.batch.core.JobExecutionListener(1)

Job

Example Listener

=>Event is an action raised on the object or comp

=>Event handling means executing some logic when the event is raised

=> Event Listener provides event handling methods.. i.e we can write event handling logic inside the event handling methods

==========

@Component("jmListener")

public JobMonitoringListener implements JobExecutionListener{

class

public void beforeJob(JobExecution

execution){

=> JobExecutionListener (1) is given to perform event handling

On the Job object creation, destruction activities..

//start time

}

JobExecutionListener(1)

public void afterJob(JobExecution execution){

gives two envent handling methods

//end time

a) beforeJob(JobExecution execution)

b) afterJob(JobExecution execution)

...

}

}

Job object

Job obj ---> Main task

Step obj ---> sub task

==========

=>It is the obejct of class that implements org.springframework.batch.core.Job(1)

=> Job defines the work to be completed.. (represents whole task to complete)

=> Generally one Application contains one Job object with 1 or more Step objects(tasks/sub tasks)

=> To create Job object we use JobBuilderFactory that comes Spring batch app through AutoConfiguration processs.

=> Job object creation needs multiple details like

(BuidlerDesign pattern)

of

name,

listener, incrementor (specifies the order executing setps),

starting step,next step, next next step, ...

logical

name

Sample code

==========

next step

JobExecutionListener (obj)

object

Job object

Incrementor

next step

object

starting Step obj

(To specify order of

executing the steps)

=>Instead of taking seperate @Configuration class

we can take place these @Bean methods in

main class where @SpringBootApplication annotation

@Configuration

@EnableBatchProcessing

public class BatchConfig {

@Autowired

is placed.

conclusion::

pre-defined

=> if there are one or two classes to configure as spring beans

then place @Bean methods in main class where @SpringBootApplication

AutoConfiuration

based Autowiring =>if there are multiple java classes to configure as spring beans then place @Bean methods in sperate @Configuration class which Autowiring is taken in the sub pkgs of @SpringBootApplication class

Direct

private StepBuilderFactory sbFactory; Through annotation is placed @Autowired private JobBuilderFactory jobFactory; @Autowired private JobExecutionListener listener; @Autowired private CustomerItemProcessor processor; //readercfg

....

@Bean method

//writer cfg @Bean method

@EnableBatchProcessing

Enables Spring Batch features and provide a base configuration for setting up batch jobs in an @Configuration class, roughly equivalent to using the <batch:*> XML

namespace.

//step cfg

@Bean(name="step1"), public Step createStep() {

return sbFactory.get("step1") //step name

.<String,Customer>chunk(10) //chunk size + Input, output types

.reader(createReader()) //reader obj

.processor(custProcessor) //processor obj

.writer(createWriter()) //writer obj

.build(); // build() method builds the step

}

// Job cfg

@Bean(name="job1")

public Job createJob1(){

return jobFactory.get("job1") // job name

of

.incrementor(new RunIdIncrementor()) // speficies the order executing given steps

.listener(listener) //specifices the listener object

method chaining

.start(createStep1()) //starting step

RunIdIncrementor

+

//.next(createStep2()) //next step

builder dp is

//.next(createStep3()) //next step

used

.build(); //builds the Job object

}

JobLauncher(1)

From

Client App

↑ implements

SimpleJobLauncher(c)

End to End Technical View /flow of Spring Batch App 2.x and 3.x

can be given from Client App

This incrementer increments a "run.id" parameter of type Long from the given job parameters. If the parameter does not exist, it will be initialized to 1. The parameter name can be configured using setKey(String).

reads from

1

implements

reader

ItemReader(1)

processor

implements mProcessor(1)

1

writer

impementWriter(1)

writes to

Transaction Manager obj JobRepository obj(3.x)

(3.x)

name

JobRepository name obj(3.x) starts the

listener

1

-start

1<1,0>

Job

Step > step

incrementor

run(job, parameters

uses

(3.x). (2.x) JobExcecutionListener(1) (Develop Impl class for

next step created using or JobBuilder StepBuilderFactory (2.x) JobBuilderFactory

chunk

size

created using

(or) StepBuilder (3.x)

to create our Listener)

JobParameters are

created

created using JobParametersBuilder obj

Client App in spring batch App

=========

=>Here we inject JobLancher object given by AutoConfiguration and also Job object cfg in

@Configuration clasusing @Autowired annotation.

=> We build JobParameters (optional) using JobParametersBuilder obj

=> we call run(job, paramters) on JobLaucher object to run the job.

```
public class Batch ProcessingTest{
@Autowired
private JobLauncher launcher;
```

based Autowiring

using AutoConfiguration

using Autowiring

```
private Job job;
@Autowired
```

Source Repository

if spring boot batch starter is added to spring app then

we get the following object through AutoConfiguration

```
ps v main(String args[]){
JobParameters params= new JobParameterBuilder()
.addLong("jobId",new Random().nextInt(10000)).toJobParameters();
launcher.run(job,params);
}//main
}//class
```

Developing the Spring Boot Batch Application using sprong boot 3.x setup

=>JobBuilderFactory

=>StepBuilderFactory

=>JobLauncher-

(optional)

Spring boot 2.x

Destination Repository

=> In spring boot batch 5.x (part of spring boot 3.x) we are not getting the following objects through AutoConfiguration

Spring boot 3.x gives following objs in batch processing through auto confiugration

a) StepBuilderFactory obj

b) JobBuilderFactory obj

2-param constructors of

=> As alternate, we need to use

StepBuilder, JobBuilder classes to create the objects manually

Step obj creation in spring boot 3.x

==========

```
// Sample with v4 (Spring boot 2.x)
@Configuration
@EnableBatch Processing
public class MyStepConfig {
@Autowired
private StepBuilderFactory stepBuilderFactory;
```

a) JobLauncher

b) JobRepository obj

c) TransactionManager obj

and etc..

=> if the Container managed AutoConfiuguration objs are required in in multiple methods @Configuration class then we need to inject them to @Configuration class by using @Autowired annotation at class level has-a properties

```
@Configuration
class AppConfig{
@Autowired
private JobRepository repository;
@Bean
public <RT> method1(){
@Bean
public Step myStep() {
return this.stepBuilderFactory.get("myStep")
.tasklet(..) // or .chunk()
.build();
}
}
// Sample with v5 (Spring boot 3.x code)
```

```
@Configuration
// @EnableBatch Processing (Not required in 3.x code)
public class MyStepConfig {
}
@Bean
}
@Bean
public <RT> method2(){
}
}
```

...

=> if the Autoconfiguration based spring bean obj is required only in @Bean method of @Configuration class then need to take the obj as the parameter of @Bean method

```
@Configuration
public class AppConfig{
@Bean
AutoConfiguration objs
public <RT> method1(Job Repository repository){
}
public Step myStep(Job Repository jobRepository, return new StepBuilder("myStep", jobRepository)
.chunk(chunkSize, transactionManager)
PlatformTransactionManager transactionManager) {
.reader(....) 10 .writer(...)
.processor(....)
.build();
}
```

Job Object creation in spring boot 3.x

==========

```
// Sample with v4
@Configuration
@EnableBatchProcessing
public class MyJobConfig {
@Autowired
private JobBuilderFactory jobBuilderFactory;
@Bean
public Job myJob(Step step) {
return this.jobBuilderFactory.get("myJob")
```

```
.start(step)
.build();
}
}
// Sample with v5
@Configuration
@EnableBatchProcessing (not required)
public class MyJobConfig {
@Bean
public Job myJob(Job Repository jobRepository, Step step) {
return new JobBuilder("myJob", jobRepository)
```

=>In spring boot 2.x, the Step,Job objects will be created using StepBuildFactory, JobBuilderFactory objects that are created through AutoConfiguration

=>In spring boot 3.x, the Step,Job objects will be created using two param constructors

of StepBuilder, JobBuilder classes

```
.start(step)
.build();
}
}
```