Spring Batch Framework

Introduction

Batch Overview

In today’s enterprise domain environment, number of business operations includes automated, complex processing of large volumes of information that is most efficiently processed without user interaction. A typical batch program generally reads a large number of records from a database, file, or queue, processes the data in some fashion, and then writes back data in a modified form. These operations may very from a simple time based events like loading or extracting data, to periodic application of complex business rules processed repetitively across very large data sets which may also requires formatting, validation and processing in a transactional manner into the system of record.

Spring Batch Overview

Spring Batch is a lightweight, comprehensive batch processing framework enhancing the development experience by enabling development of robust batch applications. This POJO-based development framework also provides reusable components that are essential in processing large volumes of records, including logging/tracing, transaction management, job processing statistics, job restart, skip, and resource management. Advance technical features available in spring enables development of high-volume and high performance batch jobs though optimization and partitioning techniques in a highly scalable manner. It is not a scheduling framework for which many good schedulers available such as Quartz, Tivoli, Control-M, etc.

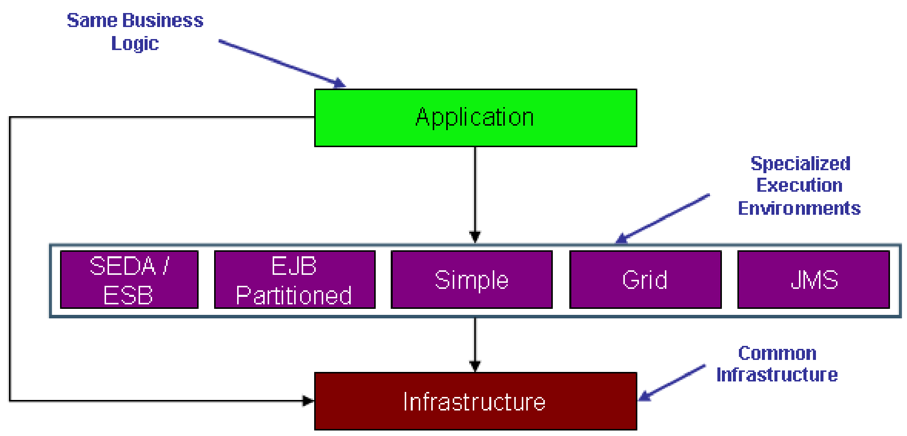
Features

1. Capability and support for multiple file formats like fixed width, delimited, xml.
2. Automatic retry after failure
3. Job control language for monitoring and operations - start, stop, suspend, cancel
4. Execution status and statistics during a run and after completion
5. Multiple ways to launch a batch job - http, Unix script, incoming message, etc.
6. Ability to run concurrently with OLTP systems
7. Ability to use multiple transaction resources
8. Support core batch services - logging, resource management, restart, skip, etc.

Architecture

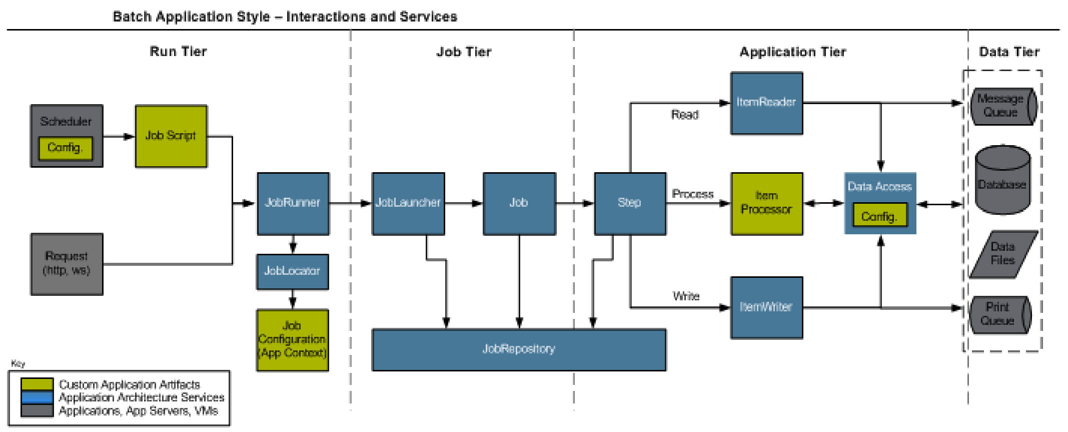
Spring batch is a three layer architecture providing a great deal of freedom to application architectures, as well as to provide batch execution environments.

Spring Batch Framework



Spring Batch provides an Infrastructure layer in the form of low level tools. There is also a simple execution environment, using the infrastructure in its implementation. The execution environment provides robust features for traceability and management of the batch lifecycle.

The Batch Application Style is organized into four logical tiers, which include Run, Job, Application, and Data. The primary goal for organizing an application according to the tiers is to embed what is known as "separation of concerns" within the system. These tiers can be conceptual but may prove effective in mapping the deployment of the artifacts onto physical components like Java runtimes and integration with data sources and targets. Effective separation of concerns results in reducing the impact of change to the system.



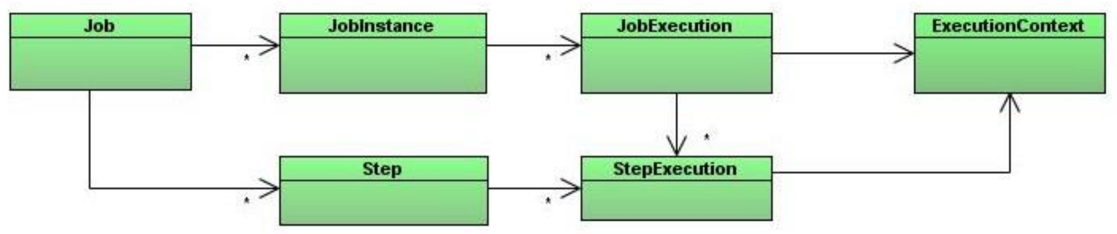
As a consequence, the conceptual tiers are:

- ***Run tier*** - responsible for scheduling and launching the application
  - used in conjunction with a scheduling product to allow time-based and interdependent scheduling of batch jobs as well as providing parallel processing capabilities
- ***Job tier*** - responsible for the overall execution of a batch job

Spring Batch Framework

- ***Application tier*** - contains specific tasks that address required batch functionality and enforces policies around execution (e.g., commit intervals, capture of statistics, etc.)
- ***Data tier*** - provides integration with the physical data sources that might include databases, files, or queues.

As seen above, the stereotypes conceptual relationship can be better depicted with the following:



Why spring batch?

The spring batch framework provides the basic feature of spring while developing batch jobs.

1. Developers will concentrate on business logic and let the framework take care of infrastructure.
2. Clear separation of concerns between the infrastructure, the batch execution environment, and the batch application.
3. Provide common, core execution services as interfaces that all projects can implement.
4. Provide simple and default implementations of the core execution interfaces that can be used ‘out of the box’.
5. Easy to configure, customize, and extend services, by leveraging the spring framework in all layers.
6. All existing core services should be easy to replace or extend, without any impact to the infrastructure layer.

Spring Batch model objects -

A batch Job is composed of one or more Steps. A JobInstance represents a given Job, parametrized with a set of typed properties called JobParameters. Each run of a JobInstance is a JobExecution. Imagine a job reading entries from a data base and generating an xml representation of it and then doing some clean-up. We have a Job composed of 2 steps: reading/writing and clean-up. If we parametrize this job by the date of the generated data then our Friday the 13th job is a JobInstance. Each time we run this instance (if a failure occurs for instance) is a JobExecution. This model gives a great flexibility regarding how jobs are launched and run. This naturally brings us to launching jobs with their job parameters, which is the responsibility of JobLauncher. Finally, various objects in the framework require a JobRepository to store runtime information related to the batch execution. In fact, Spring Batch domain model is much more elaborate but this will suffice for our purpose.

For each job, we will use a separate xml context definition file. However there is a number of common objects that we will need recurrently. All these documents will be grouped in an applicationContext.xml which will be imported from within job definitions. Let's go through these common objects:

JobLauncher

JobLaunchers are responsible for starting a Job with a given job parameters. The provided implementation, SimpleJobLauncher, relies on a TaskExecutor to launch the jobs. If no specific TaskExecutor is set then a SyncTaskExecutor is used.

JobRepository

SimpleJobRepository implementation is used which requires a set of execution Dao’s to store its information.

JobInstanceDao, JobExecutionDao, StepExecutionDao

These data access objects are used by SimpleJobRepository to store execution related information. Two sets of implementations are provided by Spring Batch: Map based (in-memory) and Jdbc based. In a real application the Jdbc variants are more suitable but we will use the simpler in-memory alternative in this example.

Tasklets

A tasklet is an object containing any custom logic to be executed as a part of a job. Tasklets are built by implementing the Tasklet interface. There is execute method which returns an ExitStatus to indicate the status of the execution of the tasklet.

Running the Job

Now we need something to kick-start the execution of our jobs. Spring Batch provides a convenient class to achieve that from the command line: CommandLineJobRunner. In its simplest form this class takes 2 arguments: the xml application context containing the job to launch and the bean id of that job. It naturally requires a JobLauncher to be configured in the application context.

```
java org.springframework.batch.core.launch.support.CommandLineJobRunner
simpleJob.xml simpleJob
```

Example

Flat File to Database

This is an effort to discuss the spring batch framework with example. This sample will read rows from a flat file and insert them into the database. The file is a comma separated file with format => notes,date.

Spring Batch Framework

Table structure –

Field	Type	Null	Key	Default	Extra
id	int(11)	No	PRIMARY	NULL	auto_increment
note	varchar(45)	No		NULL	
created_on	datetime	No		NULL	

Here is the snippet from **spring application context xml** file...

```
<!-- 3) JOB REPOSITORY - WE USE IN-MEMORY REPOSITORY FOR OUR EXAMPLE --
>
<bean id="inmemoryJobInstanceDao"

class="org.springframework.batch.core.repository.dao.MapJobInstanceDao"
/>
 <bean id="inmemoryJobExecutionDao"

class="org.springframework.batch.core.repository.dao.MapJobExecutionDao
" />
 <bean id="inmemoryStepExecutionDao"

class="org.springframework.batch.core.repository.dao.MapStepExecutionDa
o" />
<bean id="jobRepository"

class="org.springframework.batch.core.repository.support.SimpleJobRepos
itory">
      <constructor-arg ref="inmemoryJobInstanceDao" />
      <constructor-arg ref="inmemoryJobExecutionDao" />
      <constructor-arg ref="inmemoryStepExecutionDao" />
  </bean>

<!-- 4) LAUNCH JOBS FROM A REPOSITORY -->
<bean id="jobLauncher"

class="org.springframework.batch.core.launch.support.SimpleJobLauncher"
>
            <property name="jobRepository" ref="jobRepository" />
</bean>

<!-- 5) Define task step. Configure its readers and writers -->
<bean id="taskletStep"

class="org.springframework.batch.core.step.item.SimpleStepFactoryBean">
            <property name="jobRepository" ref="jobRepository" />
    <property name="itemReader" ref="reader" />
    <property name="itemWriter" ref="writer" />
    <property name="transactionManager" ref="transactionManager" />
    <property name="commitInterval" value="1000" />
</bean>

<!-- 6) Define the job and its steps. -->
<bean                                                   id="simpleJob"
class="org.springframework.batch.core.job.SimpleJob">
        <property name="name" value="simpleJob" />
```

```
          <property name="steps">
          <list>
              <ref bean="taskletStep" />
          </list>
        </property>
        <property name="jobRepository" ref="jobRepository" />
</bean>

<!-- ======================================================= -->
<!--  7) READER -->
<!-- ======================================================= -->
<bean                                                       id="inputFile"
class="org.springframework.core.io.ClassPathResource">
            <constructor-arg value="com/batch/todb/notes.txt" />
</bean>
<bean                                                          id="reader"
class="org.springframework.batch.item.file.FlatFileItemReader">
        <property name="resource" ref="inputFile" />
        <property name="firstLineIsHeader" value="false" />
        <property name="lineTokenizer">
                  <bean

      class="org.springframework.batch.item.file.transform.DelimitedLin
eTokenizer">
                <property name="delimiter" value="," />
            </bean>
        </property>
        <property name="fieldSetMapper">
            <bean class="com.batch.todb.NoteMapper" />
        </property>
  </bean>
<!-- ======================================================= -->
<!--  8) WRITER -->
<!-- ======================================================= -->
<bean id="writer" class="com.batch.todb.NoteWriter" />
```

3 and 4 – Define job launcher configuration for batch framework
5 - Defines the step and the reader and writer. In this case the reader reads from the flat file and then sends it to the writer to persist to the database.
6 - Define the job and its step.
7 - FlatFileItemReader which will read the rows from the flat file and pass it one by one to the writer to persist to the database..
8 - Configure the writer to write to the database.

Now some of the snippet from **Java code**.

```
//=======================================================
// NOTES DAO
//=======================================================
@Transactional(propagation = Propagation.REQUIRED)
public void save(final Note note) {
              super.update("insert   into   notes   (note,   created_on)
values(?,?)", new PreparedStatementSetter() {
```

Spring Batch Framework

```
                     public   void   setValues(PreparedStatement   stmt)
throws SQLException {
                           stmt.setString(1, note.getNotes());
                           stmt.setDate(2,
                                         new
java.sql.Date(note.getCreatedOn().getTime()));
                     }
              });
       }
}

//========================================================
// NOTES MAPPER
//========================================================

       public Object mapLine(FieldSet fs) {
              Note note = new Note();
              int idx = 0;
              note.setNotes(fs.readString(idx++));
              Date d = fs.readDate(idx++);
              note.setCreatedOn(d);
              return note;
       }
```

The complete code is attached at the end of the document.

Definition

Job - A job represents your entire batch work. Each night you need to collect all of the 1)credit card transactions, 2)collect them in a file and then 3)send them over to the settlement provider. Here I defined three logical steps. In Spring Batch a job is made of up of Steps. Each Step being a unit of work.

JobInstance - A running instance of the job that you have defined. Think of the Job as a class and the job instance as your , well object. Our credit card processing job runs 7 days a week at 11pm. Each executions is a JobInstance.

JobParameters - These are nothing but parameters that go into a JobInstance.

JobExecution - Every attempt to run a JobInstance results in a JobExecution. For some reasons Jan 1st, 2008 CC Settlement job failed. It is re-run and now it succeeds. So we have one JobInstance but two executions (thus two JobExecutions). There also exists the concept of StepExecution. This represents an attempt to run a Step in a Job.

JobRepository - This is the persistent store for all of our job definitions. In this example I setup the repository to use an in-memory persistent store. You can back it up with a database if you want.

JobLauncher - As the name suggests, this object lets you launch a job.

Spring Batch Framework

TaskLet - Situations where you do not have input and output processing (using readers and writers).

ItemReader - Abstraction used to represent an object that allows you to read in one object of interest that you want to process. In my credit card example it could be one card transaction retrieved from the database.

ItemWriter - Abstraction used to write out the final results of a batch. In the credit card example it could be a provider specific representation of the transaction which needs to be in a file. Maybe in XML or comma separated flat file.

ItemProcessor - Very important. Here you can initiate business logic on a just read item. Perform computations on the object and maybe calculate more fields before passing on to the writer to write out to the output file.

Reference

http://static.springsource.org/spring-batch/reference/html/index.html