D

Introducing Batch Processing

What is Batch Processing

Batch Applications enable programmers to define, implement, and execute batch jobs which are tasks that do not require human interaction for execution. Batch framework includes a job specification language usually based on XML to define batch jobs, a Java API to implement batch jobs, and a batch runtime to execute batch jobs. Many enterprise applications contain batch jobs that are executed periodically to process large amount of information such as log files, or database records. Some examples of such applications are billing report generation, and image processing.

Batch processing refers to executing batch jobs on a computer system. Java EE 7 provides a batch processing framework to support development and execution of batch applications. It includes an XML based job specification language, a set of batch annotations and interfaces to implement business logic in the application classes, a batch container to manage execution of batch jobs, and various supporting interfaces and classes that can interact with the batch container.

Batch jobs usually comprise of various sequentially executable phases which are termed as steps. For example, in a billing report generation, the first step will associate each billing entry with a billing amount, and the second step calculates and enters the total billing amount. Thus, all batch applications define a set of steps and their order of execution. The steps in a batch job can be categorized as follows:

- **Chunk-oriented steps**—Comprise of three parts viz input retrieval part that reads one item at a time from a source of data, business processing part that applies some business logic to the retrieved data and writes the result to a chunk, and output writing part that writes the chunk back to the data source. Chunk steps are long-running as they need to process large amount of data and can save their progress in the case of interruptions using checkpoints. The chunk step can then be later restarte from last checkpoint.
- **Task-oriented steps**—Execute a particular task. For example, moving files, and configuring resources.

Each step in a batch job is associated with a status to indicate if its state that whether it is running, interrupted, or completed. The steps can also have decision elements, to determine the next step or termination.

Job Specification Language

Job Specification Language helps in defining the steps and their execution order for a batch job in an XML file as shown in the following code snippet:

```
<job id="loganalysis" xmlns="http://xmlns.jcp.org/xml/ns/javaee"
version="1.0">
  <properties>
    <property name="input_file" value="inputLogs.txt"/>
    <property name="output_file" value="outputLogs.txt"/>
  </properties>
  <step id="logprocessor" next="cleanup">
    <chunk checkpoint-policy="item" item-count="10">
      <reader ref="com.kogent.logpkg.LogItemReader"></reader>
```

```
        <processor ref="com.kogent.logpkg.LogItemProcessor"></processor>
        <writer ref="com.kogent.logpkg.LogItemWriter"></writer>
      </chunk>
    </step>
    <step id="cleanup">
      <batchlet ref="com.kogent.logpkg.CleanUp"></batchlet>
      <end on="COMPLETED"/>
    </step>
  </job>
```

The code snippet shows one batch job named loganalysis, having a chunk step named logprocessor and a task step named cleanup. The logprocessor step is followed by the cleanup step, which terminates once the job is completed.

The various elements used in the code snippet to define the batch job are described as follows:

- The <job> element is always the top level element and defines the batch job
- The <properties> element helps in defining various properties and configuration parameters
- The <step> element can be the child element of <job> element and can further have following child elements:
  - One <chunk> element to define chunk-oriented step
  - One <properties> element
  - One <listener> element
  - One <end>, <stop>, and/or <fail> element.
- The <batchlet> element is a child element of <step> element and specifies a batch artefact that implements Batchlet interface.

Java API for Batch Processing

Once the jobs are defined as batch artifacts using Job Specification Language (JSL), the artifacts are created as Java classes that implement the interfaces in the javax.batch.api package and its sub-packages. The main batch artifacts interfaces are listed in Table D.1:

Table D.1: Main Batch Artifacts Interfaces

Package Name	Interface	Desription
javax.batch.api	Batchlet	Referenced from the <batchlet> element to implement business logic for a tsk-oriented step
javax.batch.api.chunk	ItemReader	Referenced from the <reader> element to read items from an input data sources
javax.batch.api.chunk	ItemProcessor	Referenced from the <processor> element to process the items received from an input data sources
javax.batch.api.chunk	ItemWriter	Referenced from the <writer> element to write output items in a chunk step

Executing Jobs in Batch Runtime

Once the jobs are defined and implemented as Java classes, they can be submitted to the batch runtime using the JobOperator interface in the javax.batch.operations package. Using the interface, you can also obtain information about the executing jobs. The JobOperator object can be obtained by using the getJobOperator factory method from the BatchRuntime class in the javax.batch.runtime package.

A job can be submitted to the batch runtime as shown in the following code snippet:

```
JobOperator jobOperator = BatchRuntime.getJobOperator();
Properties props = new Properties();
props.setProperty("parameter1", "value1");
...
long jobExecID = jobOperator.start("firstjob", props);
```

The status of the batch job can be obtained using its existing execution ID as shown in the following code snippet:

```
JobExecution jobExec = jobOperator.getJobExecution(jobExecID);
String status = jobExec.getBatchStatus().toString();
```