Batch Processing in Spring

Joseph Faisal Nusairat
Groovy Sage

Batch Processing

- What is batch processing?
- Issues associated with it?
- How do we solve it today?
- How does Spring Batch help solve it?

What is Batch Processing

Thursday, June 10, 2010

In here we discuss the batch processing issues.

Batch Processing Definition

- "Batch processing is execution of a series of programs ('jobs') on a computer without manual intervention." - Wikipedia
  - Processes can be ran without manual intervention.
  - All input data is preconfigured.
  - Data is collected into "batches".

Thursday, June 10, 2010

These are the more standard definitions for batch processing.

Benefits of Batch Processing

- Processing allows data to be ran without any manual intervention.
- It allows the timing to be processed in off hours when system load is low.
- Been around since the 1950s

We have been making use of batch processing for years.
Works very well for many of the items we perform these days.

Common Batch Solutions

How to Perform Batch Processing?

- There are multiple approaches to batch processing today with Java apps.
- The idea is still to run applications in the background and use Java services to process them.
- Generally this means to call out to Spring beans or EJBs.

Obviously there are many ways of doing this. I will only discuss two for the sake of brevity. These are common Java oriented solutions for the most part

Cron Job Method

- Create a batch unix script that runs in the background.
- Can check a folder for input or a database and when input is found calls out to a Spring bean by accessing its application context.
- Or can call out to a web service.
- This can run concurrent steps if need be.

The batch script may just run continuously or check a folder for input

Downsides

- What if this process requires multiple steps? Where do you put them?
- How do you run things multi-threaded?
- Once started how do we stop / restart in the batch processing?
- Inability to throttle the processing of the batch.
- Not easily triggered from the web.

Logic in the cron job? Logic in the Batch?
While it is possible it makes it more difficult
No way to easy stop it.

JMS Queue Method

- Very common approach in business today.
- In this approach you trigger it by having a process add many items to a message queue.
- Very easy to scale the amount of threads and throttle back.
- In addition it can easily stop processing temporarily.

This is an extremely common approach.

Downsides

- There are limits to the amount of messages in a message queue.
- No easy way to discern in an individual queue if the messages belong to which batch.
- No easy way to determine completion of a batch, making steps impossible.

Common Issues

- Counting Success? Failures? Errors?
- How do we handle restarts?

This is an extremely common approach.

Conclusion?

- We need a better way.
- The current ways are great in small scale simple post processing areas but have too many weaknesses.
- Many people will either create there own framework or now have to rely on others.

What does Spring Batch Provide?

Getting Spring Batch Up and Running.

Basics

- Provides a basis ability to create a batch job that can be ran synchronously or asynchronously.
- Each jobs are given unique ways to be identified and configured.
- Running of the batch jobs is in steps. These steps are configurable and even controllable by the flow.

Basics

- The actual services of reading and writing of the data can be handled by services that are agnostic to the concept of being in a batch job. Meaning these services are easily reusable through the application.
- The batch jobs are configurable via XML namespaces in a spring batch job.

Basics of Spring Batch

So how does spring batch help?

Setting Up Spring Batch

Thursday, June 10, 2010

Getting Spring Batch Up and Running.

Downloading Spring Batch

- Download Spring Batch from: http://static.springsource.org/spring-batch/downloads.html
- Site also contains many resources on spring batch.
- Luckily spring batch makes use of many namespaces.

Database Configuration

- Spring batches uses a database to store information about the jobs, batches, and even some statistics.
- Will have to load up database tables for Spring Batch. This will help drive database items.

Minimum Configurations

- Since Spring Batch uses a database for persisting information about the jobs we will have to configure a datasource and transaction manager

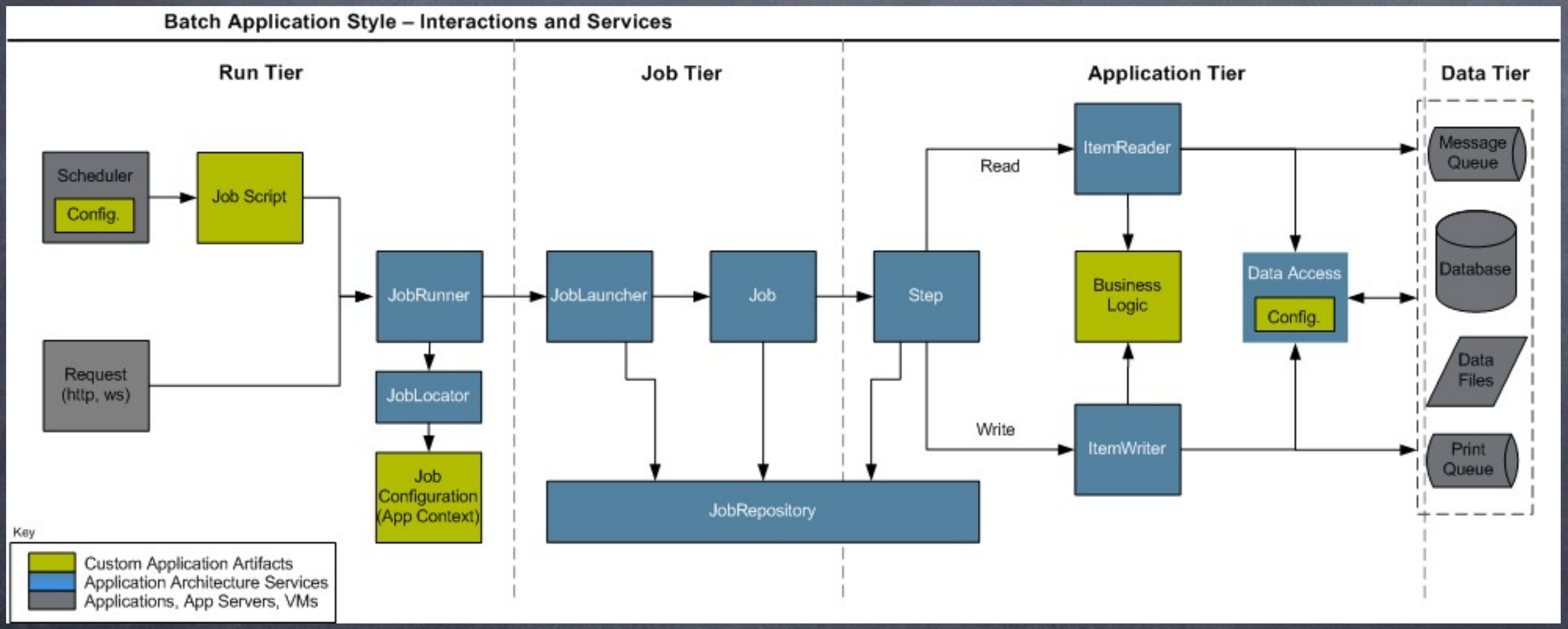
```
<bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
        <property name="jndiName">
            <value>java:/EscDatasource</value>
        </property>
</bean>

<bean id="transactionManager"
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"/>
</bean>
```

How Spring Batch Works

- Spring batch works by wrapping your services (or using built in services) in a framework that handles the usual batch items.
- This can include items used for monitoring, stopping, restarting, etc.
- All the necessary features to push data from step to step.

Spring Batch Steps



The Job

Jobs

- Spring Batch works by allowing the user to define a Job.
- One can define multiple jobs in your spring application.
- The job represents the entirety of the process you want to perform be it to process football scores or a multi step process.

Jobs are easily defined

Defining a Job

- Jobs are easily defined with XML Namespaced objects
- Jobs can also be defined with regular bean names as well.

```
<job id="footballJob”>
    <!-- steps -->
</job>
```

Steps

- Jobs are composed of one or more steps. This is where all the work is defined.
- There has to be at least one or more steps in a job.
- When you have multiple steps its necessary to define the flow to the next step. This can either be linear or conditional.
- Steps can be defined in the context of a job or outside of a job.

Creating a Linear Step

- Linear steps are just steps that go in a row.
- They each must define what there next step is with the final step having no next.
- The parent points to a predefined step.

```
<job>
  <step id="playerload" next="gameLoad">
    <!-- This is where our work will go -->
  </step>
  <step id="gameLoad" next="playerSummarization">
    <tasklet>
      <!-- This is where our work will go -->
    </tasklet>
  </step>
  <step id="playerSummarization" parent="summarizationStep" />
</job>
```

please note step ids have to be unique ACROSS ALL JOBS
The predefined step can be defined outside of the job.

Creating a Conditional Flow

- Conditional logic is used to control flow based on exit status of a step. (not to be confused with batch status)

```
<job id="job">
    <step id="stepA" parent="s1">
        <next on="*" to="stepB" />
        <next on="FAILED" to="stepC" />
    </step>
    <step id="stepB" parent="s2" next="stepC" />
    <step id="stepC" parent="s3" />
</job>
```

Possible Exit Status

- Possible exit statuses of a step:
  - COMPLETED
  - EXECUTING
  - FAILED
  - NOOP
  - STOPPED
  - UNKNOWN

Components of a Step

- Steps as we recall are where the work happens.
- This is where the services a JMS Queue used is called etc.
- The are two basic types of steps.
  - Ones that perform all the work in one action
  - Others that get the action split apart.

Tasklet

- A tasklet is a single unit of work within a step.
- This is probably a concept one is most used to seeing when creating batch processes before.
- A tasklet will simply perform one task. Whatever you want to encode in here you can.
- Often very simple business logic items like writing many items to a database or calling a stored proc.

Thursday, June 10, 2010

This is generally useful for smaller patterns and will not bode well to any multi synchronous step.
In addition unless you define transactioning yourself it could potentially be a large transaction

Defining a Tasklet

- Tasklets are easy to define in your steps.
- Simply pass the tasklet namespace a reference to the tasklet.

```
<step id="step1">
    <tasklet ref="myTasklet"/>
</step>
```

Creating a Tasklet

- There are two ways to create a tasklet.
- One is to simply use an existing service and have a tasklet adapter wrap a proxy around it.
- Another is to create your own class that implements Tasklet. This approach gives you more access to properties on the job itself.

Using the Adapter

- With the adapter simply define your own Spring component then reference it with the tasklet adapter.

```
<bean id="myTasklet"
class="org.springframework.batch.core.step.tasklet.TaskletAdapter"
>
    <property name="targetObject">
        <bean class="org.mycompany.FooDao">
    </property>
    <property name="targetMethod" value="updateFoo" />
</bean>
```

The myTasklet is then refernced ind our step. Where the method updateFoo is called.

Creating a Bean

- With this method you will create a java class that is a tasklet itself.

```
public class FileDeletingTasklet implements Tasklet {
    private Resource directory;

    public RepeatStatus execute(StepContribution contribution,
                                ChunkContext chunkContext)
        throws Exception {
         // do the work here
         return RepeatStatus.COMPLETED;
    }
}
```

You could also use InitailizingBean if you want initialize items first.
We will see later reasons to use ths

Reader / Writers

- Readers and writers are one of the core features we saw that really make Spring Batch shine.
- This ability helps to provide automated and smooth processing of data.
- Readers are designed to read the data who's results are then fed into the writer in batches.

Readers

- The reader is in charge of reading the data.
- Data can be read from any source be it a straight list, a flat file, a database, etc.
- Reading is complete when the read method returns a null object.
- Spring Batch comes with many built in readers.

```
public interface ItemReader<T> {
    T read();
}
```

Built in Readers

- ListItemReader
- AggregateItemReader
- FlatFileItemReader
- StaxEventItemReader
- JdbcCursorItemReader
- HibernateCursorItemReader
- IbatisPagingItemReader
- JmsItemReader
- JpaPagingItemReader
- JdbcPageItemReader

AbstractItemCountingItemStreamItemReader - Abstract base class that provides basic restart capabilities by counting the number of items returned from an`ItemReader`. Also an ItemReaderAdapter ... adapts any class to a reader interface

Defining a Reader

```
<bean id="playerFileItemReader"
   class="org.springframework.batch.item.file.FlatFileItemReader">
   <property name="resource" value="classpath:data/footballjob/input/${player.file.name}" />
     <property name="lineMapper">
      <bean class="org.springframework.batch.item.file.mapping.DefaultLineMapper">
        <property name="lineTokenizer">
          <bean class="org.springframework.batch.item.file.transform.DelimitedLineTokenizer">
           <property name="names"
                value="ID,lastName,firstName,position,birthYear,debutYear" />
          </bean>
        </property>
        <property name="fieldSetMapper">
          <bean class="org.spring....football.internal.PlayerFieldSetMapper" />
        </property>
       </bean>
    </property>
</bean>
```

This is how to define a reader. THis is for an out of the box one even though a custom one would look the same.

Writers

- The writers are used to process items fed into it by the readers.
- Writers receive items in a list of data.
- This allows a batch processing of the data.

Built in Writers

- CompositeItemWriter
- PropertyExtractingDelegatingItemWriter
- FlatFileItemWriter
- HibernateItemWriter
- JdbcBatchItemWriter
- JpaItemWriter
- StaxEventItemWriter

Defining a Writer

```
<bean id="fileItemWriter1" class="org.springframework.batch.item.file.FlatFileItemWriter">
    <property name="name" value="fw1" />
    <property name="resource" value="file:target/test-outputs/CustomerReport1.txt" />
    <property name="lineAggregator">
      <bean
        class="org.springframework.batch.item.file.transform.PassThroughLineAggregator" />
    </property>
</bean>
```

This defines a simple writer.

Writer Batch Execution

- List of item is passed into the writer in order to process each one.
- Assuming all in the list are successful the method ends and the transaction closes.
- If there is an exception anywhere. The items are rolled back.
- Each item is then run individually.

This ability of controlling the transaction for a group of items allows you nto to flush every single time

Processor

- Readers feed data to the writers straight without any validation and its the same component.
- Sometimes there needs to be validation that the data is ok or the data received from the reader needs to be transformed to a different object.
- The processor is an optional component to include in the step

Defining a Processor

- The processor takes in an object the same type that the writer returns.
- The process() then returns the same method that the writer list accepts.
- These types may or may not be the same.

```
public interface ItemProcessor<I, O> {

    O process(I item) throws Exception;
}
```

Thursday, June 10, 2010

There is also the ability to chain the readers adn writers
This works great with out of the box readers/writers. Not as much for custom ones.

Integrating Readers / Writers

- After we have specified our readers and writer beans we need to integrate them into our steps.
- We define in the bean a reference to each object as well as commit-interval; this variable is the size of the list past to the rider

```
<job id="ioSampleJob">
    <step name="step1">
      <tasklet>
         <chunk reader="fooReader" processor="fooProcessor"
          writer="barWriter" commit-interval="2"/>
      </tasklet>
    </step>
</job>
```

Steps

- BAtches are composed of one or more steps.
- Each step will have a reader and writer portion or a tasklet.
- A step will end and one will move on to the next step once the step is complete.

Launching the Job

Launching the Job

- After we have created a job we are going to have to launch it.
- Launching can still occur from anywhere be it a JMS queue, Cron job, it all depends what mechanism you want to start the job.
- So the actual mechanism to start a job is up to you.

Components Needed to Start

- In order to start a job you will need to have two basic things.
  - The Job you want
  - Unique Parameters for that Job
    - The parameters should be easily rememberable parameters as well.

Thursday, June 10, 2010

The first part is obvious. SInce we can have multiple jobs we need multiple paramters
Second part isnt as obvious at first

Job Parameters

- When selecting parameters make it so its easily retrievable from the front end.
- Also consider HOW you want to retrieve them. Sometimes having a date in there is good, other times its not.
- Also consider you can restart queries as well.

Further discussion on them

To Launch a Job

- Once you have your Job and the Parameters it's easy to launch the job.

```
// Retrieve a convert to get the job
JobParametersConverter converter = new DefaultJobParametersConverter();
final Properties props = new Properties();
props.put("runId", runId);

// launch the job
JobExecution jobInfo = jobLauncher.run(job, converter.getJobParameters(props));

BatchStatus batchJobStatus = jobInfo.getStatus();
```

When the job is execute a new jobInstance is created.

Exception Conditions

- JobExecutionAlreadyRunningException
  - This occurs because u launched a job with the same parameters and is running.
- JobInstanceAlreadyCompleteException
  - Same issue but now its complete.
- JobRestartException
  - Attempting to restart a job.

These are possible exceptions from a job

Batch Statuses

- ABANDONED
- COMPLETED
- FAILED
- STARTED
- STARTING
- STOPPED
- STOPPING
- UNKWON

Stopping a Job

- Once a job has started it may be necessary to stop it.
- Please note there is no pause and stop/restart is te closest to a pause
- You will just need access to the execution Id.

```
JobOperator jobOperator;
long executionIdLong = Long.parseLong(executionId);
jobOperator.stop(executionIdLong);
```

Restarting a Job

- Jobs can only be restarted that are not in a complete status.
- The job will restart at the last step that was NOT fully completed.
  - If necessary a cleanup step can occur in the afterStep (via the StepListener) checking for STOPPED or FAILED status.

Thursday, June 10, 2010

ITs important to remember the fully complete so that there is a cleanup (we will get to step listeners in a bit)

Beyond the Basics

In here we will investigate what happens when there are errors in the step itself.

Errors in the Step

In here we will investigate what happens when there are errors in the step itself.

Errors in the Step

- Errors can occur during the read/write process.
- These can be normal errors that occur due to business processes failing.
- Errors can also occur to SQL dead locks.
- Errors can occur due to external resources not being available at the time

Allowing Errors to be Skipped

- At times it may “ok” to have a read-write step fail.
- One can configure a skippable step and the amount of times we will allow it to be skipped

```
<chunk ... skip-limit=”10”>
  <skippable-exception-classes>
    <include class="java.lang.Exception"/>
    <exclude class="java.io.FileNotFoundException"/>
  </skippable-exception-classes>
</ chunk>
```

Thursday, June 10, 2010

There could be many reasons for this from the step isnt that important or that it is simply part of the process. the client sends you bad data

Allowing Errors to be Retried

- In other situations deadlocks may occur due to simultaneous writes to the database from outside sources. Or a web service could be temporarily down. In these case you may want to retry

```
<chunk ... retry-limit="10">
  <retryable-exception-classes>
    <include class="java.lang.Exception"/>
    <exclude class="java.io.FileNotFoundException"/>
  </retryable-exception-classes>
</ chunk>
```

Processing Errors on Writer

- If you need to do any processing of the skips inside of the readers / writers this can be handled by implementing SkipListener on the class.

```
void onSkipInProcess(T item, Throwable t)
void onSkipInRead(Throwable t)
void onSkipInWrite(S item, Throwable t)
```

- One needs to define all the methods but you only need to make the one work that you need to work.

Step Listener;s

- When running a reader and writer, it is sometimes necessary to initialize a global variable.

```
public class CustomReader implements ItemReader<MyBean>,
  StepExecutionListener {

   public MyBean read() {
      return null; }

   public ExitStatus afterStep(StepExecution arg0) { }

   public void beforeStep(StepExecution stepExecution) {
     String runId = stepExecution.getJobParameters().getString("runId");
   }
}
```

Thursday, June 10, 2010

These variables may be needed to initialize variables based on the job parameters

Asynchronous Processing

In here we will investigate what happens when there are errors in the step itself.

Issues with Asynchronousity

- One of the features many people need when performing batch processing is speed.
- The easiest way to run batch processing as fast as possible is to have multiple threads working on the data.
- In most advance applications the writers will perform business and extra database logic and take time.

However there are issues ....

Support in Spring Batch

- Unfortunately Spring Batch doesn't give us 100% support we'd like to have.
- The Readers that come out of the box are NOT thread safe.
- We will have to create our own custom readers.
- We will also have to perform some customization on our steps as well.

So we have to create our own thread safe
This is not an easy task but not a hard task item either

Custom Reader

```
public class FootballTeamReader implements ItemReader<Team>, StepExecutionListener {

  private List<Team> results;
  private TeamDAO  teamDao;

  public Team read() {
   synchronized(results)) {
     if (! results.isEmpty()) {
       try {
         return (results.remove(0));
     } catch(final IndexOutOfBoundsException e) { }
   }
   return null;
  }

  public void beforeStep(StepExecution stepExecution) {
    results = Collections.synchronizedList(teamDao.getAllTeams());
  }
}
```

This is the type of code we need.
Notice the synchronization

Defining the Step

- First we create a step bean

```
<bean id="skipLimitStep"
  class="org.springframework.batch.core.step.item.SimpleStepFactoryBean"
        abstract="true">
        <property name="transactionManager" ref="transactionManager" />
        <property name="jobRepository" ref="jobRepository" />
        <property name="startLimit" value="100" />
        <property name="commitInterval" value="1" />
</bean>
```

- Please note if you are doing any of the skippable step configurations you needed to extend FaultTolerantStepFactoyBean instead.

Making it Asynchronous

- In order to make it asynchronous we are going to be making use of a core spring class that will create new multiple threads for each object.
- The parent is the bean we defined earlier.
- Concurrency limit is the amount of threads.

```
<bean id="multiThreadedStep" parent="skipLimitStep">
  <property name="taskExecutor">
    <bean class="org.springframework.core.task.SimpleAsyncTaskExecutor">
      <property name="concurrencyLimit" value="5"/>
    </bean>
  </property>
</bean>
```

Please note hte SimpleAsyncTaskExecutor will make a new thread for each item.
ThreadPooledTaskExecutor is in theory better but we had issues with it in spring batch.
It would keep running even when the ItemReaders were returning null.

Step Scope

- Most of the Spring batch components are in the prototype scope. This allows for maximum reuse of the objects.
- However when using the multi-threaded steps and having objects that need to persist across the step (any global variables) we will have to set the reader / writers scope to “step”

Accessing Job Data

In here we will investigate what happens when there are errors in the step itself.

Accessing the Jobs

- There are lots of great items in the database to monitor jobs.
- We can monitor the amount of jobs being ran, the amount failed, how many jobs have been restarted.
- In addition during the run one could even monitor the amount of items processed, skipped, etc.

Job Instance Table

- Serves at the top of the hierarchy of the jobs
- The batch_job_instance table contains:
  job_instance_id
  version
  job_name
  job_key

Batch Job Params Table

- Contains all the parameters for a job. If you can easily reference the id it will make looking up the job instances easier.
- The batch_job_params table contains:
  job_instance_id
  key_name
  string_val
  date_val
  long_val
  double_val

This is why in general i recommend a PK of a table id and let that tabel contain any configuration etc data
.. this is an abbreviated list

Batch Job Execution Table

- Contains all the information relevant to a given job execution object.
- The batch_job_params table contains:
  job_execution_id (pk)
  job_instance_id
  create_time
  start_time
  end_time
  status
  exit_code
  exit_message

.. this is an abbreviated list

Batch Step Execution Table

- Contains all the information relevant to a given step in the execution of the job.
- The batch_step_execution table contains:
  step_execution_id (pk)
  job_execution_id
  start_time
  end_time
  status
  commit_count
  read_count
  write_count
  read_skip_count
  process_skip_count

.. this is an abbreviated list

Q & A

TDD Grails Tek Spike

- Grail TDD one day camp
  - July 17, 2010
  - Will learn Grails via Test Driven Development, our testing tool of choice will be Spock
  - $50
  - **www.tekspike.com**

.. this is an abbreviated list