(https://facebook.com/keyholesoftware)

(https://twitter.com/keyholesoftware)

(https://www.linkedin.com/company/keyhole-software)

(https://www.youtube.com/channel/UCAIUNkXmnAPgLWnqUDpU

(https://keyholesoftware.com/feed)

H

Generating Large Excel Files Using Spring Batch, Part Three

JONNY HACKETT (HTTPS://KEYHOLESOFTWARE.COM/AUTHOR/JHACKETT/) /
NOVEMBER 12, 2012 /
JAVA (HTTPS://KEYHOLESOFTWARE.COM/CATEGORY/TECHNOLOGY-SNAPSHOT/JAVA-TOOLS/),
SPRING (HTTPS://KEYHOLESOFTWARE.COM/CATEGORY/TECHNOLOGY-SNAPSHOT/JAVA-TOOLS/SPR
SPRING BATCH (HTTPS://KEYHOLESOFTWARE.COM/CATEGORY/TECHNOLOGY-SNAPSHOT/JAVA-TOO
TECHNOLOGY SNAPSHOT (HTTPS://KEYHOLESOFTWARE.COM/CATEGORY/TECHNOLOGY-SNAPSHO
13 COMMENTS (HTTPS://KEYHOLESOFTWARE.COM/2012/11/12/GENERATING-LARGE-EXCEL-FILES-

While working for a client recently, I was given a small project to produce a report that would help reconcile differences in data that existed in four to five different database sources. The requirements specified a need to compare roughly 40 fields

from each of these sources against each other, and to report the differences in MS Excel format, which included details regarding how the data should be displayed in the spreadsheet.

As it turned out, the challenge was not about the amount of data being processed as I originally had suspected. Instead, the challenge became how to create a potentially large Excel file without causing memory meltdown on the server hardware.

Hopefully by sharing my experience here, it might save a little time for someone else, and thus give back a little bit to the developer community.

If you haven't read the previous posts in this series (Introducing Spring Batch (https://keyholesoftware.com/2012/06/22/introducing-spring-batch/) and Getting Started With Spring Batch (https://keyholesoftware.com/2012/06/25/getting-started-with-spring-batch-part-two/)), they serve as a quick start guide and simple example for learning the basics of Spring Batch. They also serve as the starting point for this article's example code.

The Process

For our input data to this job, we're going to be reading in the data provided from the following URL. It will generate a list of the NYSE traded stock data in CSV format. You can also click this link to download a physical file in order to take a look at the data format so you can see what to expect: http://www.nasdaq.com/screening/companies-by-name.aspx?letter=0&exchange=nasdaq&render=download (http://www.nasdaq.com/screening/companies-by-name.aspx?letter=0&exchange=nasdaq&render=download)

> **NOTE:** If using Internet Explorer, you'll need to visit the following URL: http://www.nasdaq.com/screening/companies-by-industry.aspx?exchange=NYSE (http://www.nasdaq.com/screening/companies-by-industry.aspx?exchange=NYSE) and look for the CSV download link at the top of the data results. Click the link "Download this list" and there may be a popup window in which you have to enter some text in order to get the download. This doesn't appear to be an issue with Google Chrome or Firefox, nor is it an issue reading the download in Spring Batch as an input resource.

The reader for this step will be set up almost identically to the reader example in Part Two's Getting Started With Spring Batch (https://keyholesoftware.com/2012/06/25/getting-started-with-spring-batch-part-two/), because it is a CSV file in which we're specifying a URL as the resource (and not an actual physical file that we are reading in). The only difference between this example and the configuration in Part 2's example is that we need to define a custom FieldSetMapper specific to the type of data we are mapping from the input file.

Below are the bean configurations required to set up the reader for the step to convert the incoming file from CSV into Excel format. If you downloaded the stock data file and examined its contents, you should have noticed that the first line contains header information that we should not be mapping to a data object. This header information is skipped by adding the "linesToSkip" property to the **FlatFileItemReader** bean definition as you see below:

```
<bean name="stockDataReader"
    class="org.springframework.batch.item.file.FlatFileItem
    <property name="resource"
        value="http://www.nasdaq.com/screening/companies-by
    <property name="lineMapper" ref="stockDataLineMapper" /
    <property name="linesToSkip" value="1" />
</bean>

<bean name="stockDataLineMapper"
    class="org.springframework.batch.item.file.mapping.Defa
    <property name="fieldSetMapper" ref="stockDataFieldMapp
    <property name="lineTokenizer" ref="stockDataLineTokeni
</bean>

<bean name="stockDataLineTokenizer"      class="org.springf
```

Secondly, we need to create the data object that we will map the incoming file record to. For this particular file, it will look like this:

```java
package com.keyhole.example.poi;

import java.io.Serializable;
import java.math.BigDecimal;

public class StockData implements Serializable {

    private static final long serialVersionUID = 4383231542
    private String symbol;
    private String name;
    private BigDecimal lastSale;
    private BigDecimal marketCap;
    private String adrTso;
    private String ipoYear;
    private String sector;
    private String industry;
    private String summaryUrl;

    // getters and setters removed for brevity

}
```

Now that we have defined the data object that our file will be mapped to, we need to create the custom **FieldSetMapper** implementation. It should look like this:

```java
package com.keyhole.example.poi;

import java.math.BigDecimal;

import org.springframework.batch.item.file.mapping.FieldSet
import org.springframework.batch.item.file.transform.FieldS
import org.springframework.stereotype.Component;
import org.springframework.validation.BindException;

@Component("stockDataFieldMapper")
public class StockDataFieldSetMapper implements FieldSetMap

    public StockData mapFieldSet(FieldSet fieldSet) throws
        StockData data = new StockData();
        data.setSymbol(fieldSet.readString(0));
        data.setName(fieldSet.readString(1));

        String lastSaleVal = fieldSet.readString(2);
        if ("n/a".equals(lastSaleVal)) {
            data.setLastSale(BigDecimal.ZERO);
        } else {
            data.setLastSale(new BigDecimal(lastSaleVal));
        }

        data.setMarketCap(fieldSet.readBigDecimal(3));
        data.setAdrTso(fieldSet.readString(4));
        data.setIpoYear(fieldSet.readString(5));
        data.setSector(fieldSet.readString(6));
        data.setIndustry(fieldSet.readString(7));
        data.setSummaryUrl(fieldSet.readString(8));
        return data;
    }
}
```

After defining the configuration and implementation of reading the stock data file, we're ready to move on to implementing our Excel **ItemWriter.** The two most commonly used open source Java APIs are Apache POI (http://poi.apache.org/index.html)and JExcelAPI (http://jexcelapi.sourceforge.net/). As most people might attest, generating large files typically result in a high memory footprint, as they require building the entire Excel workbook in memory prior to writing out the file.

However, beginning with Apache POI version 3.8-beta3 in June of 2011, **developers now have the option to use a low-memory footprint Excel API.** Apache POI also has additional advantages in that it is continually evolving and has a strong development community, ensuring that it will be maintained for the foreseeable future.

- If you are using Maven and an Eclipse-based IDE like SpringSource Tool Suite (STS), it's very simple to obtain the Apache POI API for your project. Setting up a Spring Batch project in STS was detailed in Getting Started With Spring Batch (http://keyholesoftware.wordpress.com/2012/06/25/getting-started-with-spring-batch-part-two/) so we won't go into detail regarding project setup. Right click on the project in STS, select "Maven" and then select "Add Dependency." In the dialogue box for the search entry, you'll want to enter POI and look for the result that corresponds to the org.apache.poi package. After that, you'll need to do the same process for poi-ooxml, likewise selecting the result that corresponds to the org.apache.poi package.
- If you are not using Maven, you'll need to visit the Apache POI (http://poi.apache.org/) website to manually download the latest version and move the required jars into your lib directory manually. This will include the poi jar, poi-ooxml jar and all of its associated jars. Details of each can be found on the Apache website (http://www.apache.org/).

See Also: Fluent Assertions with AssertJ

(https://keyholesoftware.com/2018/03/12/fluent-assertions-with-assertj/)

This new Excel API from Apache is named **SXSSF.** It is an API-compatible streaming extension of XSSF, which is used to create newer Excel 2007-based OOXML (.xlsx) files. It achieves this by limiting access to the number of rows in memory within a sliding window. For example: if you define the sliding window as 50, when the 51st

row is created, the first row that is in the window is written to disk. This operation repeats as each new row is created and the oldest row in the window is written to disk. The older rows (that are no longer in the window) become inaccessible since they have been flushed from memory.

To begin using this streaming version SXSSF, it is really just as simple as this:

```
Workbook wb = new SXSSFWorkbook(100);
```

By instantiating an **SXXFWorkbook** object and calling the constructor that accepts an integer as the parameter, we have defined our workbook for streaming with a sliding window of 100 rows.

Since the goal of this example is to reduce the memory footprint, we're going to be processing the file in chunks of 500. In order to process the file in chunks like this, we'll need to create our Excel workbook once at the beginning of the step and close the output stream at the very end of the step, while writing the data out in between. To do this, we're going to create our **ItemWriter** with methods to be processed before the step and after the step, and implemented using Spring Batch's built-in annotations.

First, here's the method that will be created to handle the BeforeStep interception. I have left out the details of how the title and header information were created, but they will be included in the complete code listing near the end.

```
1  @BeforeStep
2      public void beforeStep(StepExecution stepExecution) {
3
4          String dateTime = DateFormatUtils.format(Calendar.g
5                  "yyyyMMdd_HHmmss");
6          outputFilename = FILE_NAME + "_" + dateTime + ".xls
7
8          workbook = new SXSSFWorkbook(100);
9          Sheet sheet = workbook.createSheet("Testing");
10         sheet.createFreezePane(0, 3, 0, 3);
11         sheet.setDefaultColumnWidth(20);
12
13         addTitleToSheet(sheet);
14         currRow++;
15         addHeaders(sheet);
16         initDataStyle();
17
18     }
```

The method can be named anything, but by convention I normally name the method **BeforeStep** just to stay consistent with the purpose and use of the method. By annotating that method with @BeforeStep, this tells Spring Batch that before the

step is executed, this method should be called and the **StepExecution** passed in as a parameter. Typically these methods are used to configure resources that will be used by the bean, whether that bean is a reader, processor, writer or tasklet.

It's also important to note that if your bean is extending one or more classes, then there can only be one @BeforeStep or @AfterStep annotated method. The code listing here shows that we're defining the file name and instantiating the workbook with a row sliding window of 100 (which is the same as the default but listed here to show how that would be defined). We also need to create the first sheet, add a title / header info to that sheet, and initialize the cell style that will be used for the data output.

Here is the method on the writer that will be called in the AfterStep phase of job execution:

```
1  @AfterStep
2      public void afterStep(StepExecution stepExecution) throw
3          FileOutputStream fos = new FileOutputStream(outputFi
4          workbook.write(fos);
5          fos.close();
6      }
```

Just as with the BeforeStep annotated method, this AfterStep annotated method is typically used to wrap up necessary items after a step has completed. Just as their names imply, these methods are called before the step begins to execute and after the step has completed executing. The code listed here will create the output stream necessary for the Excel workbook to write to. And, once that has completed, we need to close the output stream. What's important to note here is that when we are calling workbook.write(fos) at this point, it's taking the temp files that were used to stream the Excel data out to disk and assembling them back into an Excel .xlsx file.

So, now that we've defined setting up the Excel workbook and closing it out, it's time to take care of the method that actually takes the data that was read from the input source and converts it into the rows and cells that will make up the detailed data of the Excel file.

Here's the code listing of **Write** method:

```
@Override
    public void write(List<? extends StockData> items) thro

        Sheet sheet = workbook.getSheetAt(0);

        for (StockData data : items) {
            for (int i = 0; i < 300; i++) {
                currRow++;
                Row row = sheet.createRow(currRow);
                createStringCell(row, data.getSymbol(), 0);
                createStringCell(row, data.getName(), 1);
                createNumericCell(row,
data.getLastSale().doubleValue(), 2);
                createNumericCell(row,
data.getMarketCap().doubleValue(), 3);
                createStringCell(row, data.getAdrTso(), 4);
                createStringCell(row, data.getIpoYear(), 5)
                createStringCell(row, data.getSector(), 6);
                createStringCell(row, data.getIndustry(), 7
                createStringCell(row, data.getSummaryUrl(),
            }
        }
    }
```

In this Write method, the code is pretty straightforward and simple. As we are looping through the list of StockData objects that were mapped from our input file, we are creating a new row and its cells for each item of data. Since the input file is only a little more than a couple of thousand rows, this wouldn't be a good test of generating a large Excel file. That's why you see the additional loop that will create 300 rows for each of the items we're going to write. By the time the job finishes, we will have generated an Excel file that has a little over 800,000 rows — **just to prove we can do it, not that you should.**

See Also: Dockerized NetScaler Web Logging (NSWL) Tool

(https://keyholesoftware.com/2017/11/28/dockerized-netscaler-web-logging-nswl-tool/)

The two methods below are convenience methods for the actual creation of each individual cell within the row to simplify some repeated code:

```
private void createStringCell(Row row, String val, int col)
    Cell cell = row.createCell(col);
    cell.setCellType(Cell.CELL_TYPE_STRING);
    cell.setCellValue(val);
}

private void createNumericCell(Row row, Double val, int col
    Cell cell = row.createCell(col);
    cell.setCellType(Cell.CELL_TYPE_NUMERIC);
    cell.setCellValue(val);
}
```

Putting it all together, here is the complete code listing for the **StockDataExcelWriter**. One important note regarding this class is its use of the **@Scope** ("step") Spring annotation. By default, Spring beans are created as singletons when they are loaded into the Spring context. Since we are holding on to state with a few items (such as the current row being written, the workbook object, and a re-usable cell style), we need the framework to instantiate this StockDataExcelWriter as needed, once per step execution. Otherwise, we could potentially run into some thread-safe issues if this job were to run simultaneously.

```
package com.keyhole.example.poi;

import java.io.FileOutputStream;
import java.io.IOException;
import java.util.Calendar;
import java.util.List;

import org.apache.commons.lang3.time.DateFormatUtils;
import org.apache.poi.ss.usermodel.Cell;
import org.apache.poi.ss.usermodel.CellStyle;
import org.apache.poi.ss.usermodel.Font;
import org.apache.poi.ss.usermodel.Row;
import org.apache.poi.ss.usermodel.Sheet;
import org.apache.poi.ss.usermodel.Workbook;
import org.apache.poi.ss.util.CellRangeAddress;
import org.apache.poi.xssf.streaming.SXSSFWorkbook;
import org.springframework.batch.core.StepExecution;
import org.springframework.batch.core.annotation.AfterStep
import org.springframework.batch.core.annotation.BeforeSte
import org.springframework.batch.item.ItemWriter;
import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;

@Component("stockDataExcelWriter")
@Scope("step")
public class StockDataExcelWriter implements ItemWriter<St

    private static final String FILE_NAME = "/data/example
    private static final String[] HEADERS = { "Symbol", "N
            "Market Cap", "ADR TSO", "IPO Year", "Sector",
            "Summary URL" };

    private String outputFilename;
    private Workbook workbook;
```

```
35      private CellStyle dataCellStyle;
36      private int currRow = 0;
37
38      private void addHeaders(Sheet sheet) {
39
40          Workbook wb = sheet.getWorkbook();
41
42          CellStyle style = wb.createCellStyle();
43          Font font = wb.createFont();
44
45          font.setFontHeightInPoints((short) 10);
46          font.setFontName("Arial");
47          font.setBoldweight(Font.BOLDWEIGHT_BOLD);
48          style.setAlignment(CellStyle.ALIGN_CENTER);
49          style.setFont(font);
50
51          Row row = sheet.createRow(2);
52          int col = 0;
53
54          for (String header : HEADERS) {
55              Cell cell = row.createCell(col);
56              cell.setCellValue(header);
57              cell.setCellStyle(style);
58              col++;
59          }
60          currRow++;
61      }
62
63      private void addTitleToSheet(Sheet sheet) {
64
65          Workbook wb = sheet.getWorkbook();
66
67          CellStyle style = wb.createCellStyle();
68          Font font = wb.createFont();
69
70          font.setFontHeightInPoints((short) 14);
71          font.setFontName("Arial");
72          font.setBoldweight(Font.BOLDWEIGHT_BOLD);
73          style.setAlignment(CellStyle.ALIGN_CENTER);
74          style.setFont(font);
75
76          Row row = sheet.createRow(currRow);
77          row.setHeightInPoints(16);
78
79          String currDate = DateFormatUtils.format(Calendar.
80                  DateFormatUtils.ISO_DATETIME_FORMAT.getPat
81
82          Cell cell = row.createCell(0, Cell.CELL_TYPE_STRIN
83          cell.setCellValue("Stock Data as of " + currDate);
84          cell.setCellStyle(style);
85
86          CellRangeAddress range = new CellRangeAddress(0, 0
87          sheet.addMergedRegion(range);
88          currRow++;
89
90      }
91
92      @AfterStep
93      public void afterStep(StepExecution stepExecution) thr
94          FileOutputStream fos = new FileOutputStream(outputFile);
```

```
95          workbook.write(fos);
96          fos.close();
97      }
98
99      @BeforeStep
100     public void beforeStep(StepExecution stepExecution) {
101         System.out.println("Calling beforeStep");
102
103         String dateTime = DateFormatUtils.format(Calendar.
104                 "yyyyMMdd_HHmmss");
105         outputFilename = FILE_NAME + "_" + dateTime + ".xl
106
107         workbook = new SXSSFWorkbook(100);
108         Sheet sheet = workbook.createSheet("Testing");
109         sheet.createFreezePane(0, 3, 0, 3);
110         sheet.setDefaultColumnWidth(20);
111
112         addTitleToSheet(sheet);
113         currRow++;
114         addHeaders(sheet);
115         initDataStyle();
116
117     }
118
119     private void initDataStyle() {
120         dataCellStyle = workbook.createCellStyle();
121         Font font = workbook.createFont();
122
123         font.setFontHeightInPoints((short) 10);
124         font.setFontName("Arial");
125         dataCellStyle.setAlignment(CellStyle.ALIGN_LEFT);
126         dataCellStyle.setFont(font);
127     }
128
129     @Override
130     public void write(List<? extends StockData> items) thr
131
132         Sheet sheet = workbook.getSheetAt(0);
133
134         for (StockData data : items) {
135             for (int i = 0; i < 300; i++) {
136                 currRow++;
137                 Row row = sheet.createRow(currRow);
138                 createStringCell(row, data.getSymbol(), 0)
139                 createStringCell(row, data.getName(), 1);
140                 createNumericCell(row, data.getLastSale().
141                 createNumericCell(row, data.getMarketCap()
142                 createStringCell(row, data.getAdrTso(), 4)
143                 createStringCell(row, data.getIpoYear(), 5
144                 createStringCell(row, data.getSector(), 6)
145                 createStringCell(row, data.getIndustry(),
146                 createStringCell(row, data.getSummaryUrl()
147             }
148         }
149     }
150
151     private void createStringCell(Row row, String val, int
152         Cell cell = row.createCell(col);
153         cell.setCellType(Cell.CELL_TYPE_STRING);
154         cell.setCellValue(val);
```


```
155         }
156
157         private void createNumericCell(Row row, Double val, in
158             Cell cell = row.createCell(col);
159             cell.setCellType(Cell.CELL_TYPE_NUMERIC);
160             cell.setCellValue(val);
161         }
162
163     }
```

Here's the Spring Batch configuration for the job:

```
1   <batch:job id="PoiExcelConverter">
2       <batch:step id="convertDataToExcel">
3           <batch:tasklet transaction-manager="transactionManag
4   <batch:chunk reader="stockDataReader"
5   writer="stockDataExcelWriter"
6   commit-interval="500" />
7           </batch:tasklet>
8       </batch:step>
9   </batch:job>
```

And now that we have proven that we can create huge Excel files, there are a few limitations to this approach as listed on the Apache POI website (http://poi.apache.org/index.html):

- Only a limited number of rows are available at a point in time, which are the rows that remain in the window and haven't been written to disk yet.
- Sheet.clone() is not supported.
- Formula evaluation is not supported.

Troubleshooting

There is one issue that I came across that took me a little while to resolve. **If you use the OOXML formats with POI you might come across this error:**

> "Excel found unreadable content in 'PoiTest.xlsx'. Do you want to recover the contents of this workbook? If you trust the source of this workbook, click Yes."

And upon clicking “Yes,” it is followed up by an error dialogue similar to this:

This usually means that you have made a mistake in defining a style somewhere in your code. By clicking the link to the log at the bottom, there’s a good chance you’ll get pointed in the right direction of where the issue is. **Hopefully this little nugget of information will save you a little time researching the error.**

Conclusion

So now that we’re done, **we have shown that there is a viable way of generating extremely large Excel workbooks without bringing the server to its knees.**

The real question now becomes: do you really need this in Excel format? Does this 800,000+ row workbook provide any real value to the business? Just because you can, doesn’t always mean that you should. **But sometimes it’s just fun to find out if you can.**

— Jonny Hackett, asktheteam@keyholesoftware.com

Spring Batch Blog Series

Part One: Introducing Spring Batch (/2012/06/22/introducing-spring-batch/)

Part Two: Getting Started With Spring Batch (/2012/06/25/getting-started-with-spring-batch-part-two/)

Part Three: Generating Large Excel Files Using Spring Batch (/2012/11/12/generating-large-excel-files-using-spring-batch-part-three/)

Scaling Spring Batch – Step Partitioning (https://keyholesoftware.com/2013/12/09/spring-batch-partitioning/)

Spring Batch Unit Testing and Mockito (https://keyholesoftware.com/2012/02/23/spring-batch-unit-testing-and-mockito/)

Spring Batch – Replacing XML Job Configuration With JavaConfig (https://keyholesoftware.com/2015/06/29/spring-batch-javaconfig/)

References

Apache POI (Excel): http://poi.apache.org/spreadsheet/index.html (http://poi.apache.org/spreadsheet/index.html)

JExcel API: http://jexcelapi.sourceforge.net (http://jexcelapi.sourceforge.net/)

Spring Batch: http://static.springsource.org/spring-batch/ (http://static.springsource.org/spring-batch/)

Recent Posts

Refreshing Your Scrum
OCTOBER 29, 2018
(https://keyholesoftware.com/2018/10/2
your-scrum/)

Topics We Write About

Microservices (https://keyholesoftware.com/category/

Blockchain (https://keyholesoftware.com/category/

JavaScript (https://keyholesoftware.com/category/snapshot/javascript/)

React (https://keyholesoftware.com/category/snapshot/javascript/react/)

Privacy Policy

Interactive REST API Documentati…
OCTOBER 24, 2018
(https://keyholesoftware.com/2018/10/2
rest-api-documentation-with-swagger-
ui/)

Release: Hyperledger Blockchain …
OCTOBER 19, 2018
(https://keyholesoftware.com/2018/10/1
hyperledger-blockchain-analytics-
tool/)

Performing Technical Interviews F…
OCTOBER 16, 2018
(https://keyholesoftware.com/2018/10/1
technical-interviews-for-consulting-
clients/)

Java
(https://keyholesoftware.com/category/
snapshot/java-tools/)

.NET
(https://keyholesoftware.com/category/
snapshot/net/)

Azure
(https://keyholesoftware.com/category/
snapshot/cloud/azure/)

AWS
(https://keyholesoftware.com/category/
snapshot/cloud/aws/)

APACHE (HTTPS://KEYHOLESOFTWARE.COM/TAG/APACHE/)

EXCEL (HTTPS://KEYHOLESOFTWARE.COM/TAG/EXCEL/)

JAVA (HTTPS://KEYHOLESOFTWARE.COM/TAG/JAVA/)

SPRING (HTTPS://KEYHOLESOFTWARE.COM/TAG/SPRING/)

SPRING BATCH (HTTPS://KEYHOLESOFTWARE.COM/TAG/SPRING-BATCH/)

TUTORIAL (HTTPS://KEYHOLESOFTWARE.COM/TAG/TUTORIAL/)

Comments 13

Pingback: Introducing Spring Batch, Part One « Keyhole Software
(http://keyholesoftware.wordpress.com/2012/06/22/introducing-spring-batch/)

Pingback: Getting Started With Spring Batch, Part Two « Keyhole Software
(http://keyholesoftware.wordpress.com/2012/06/25/getting-started-with-spring-batch-part-
two/)

Rahul Saini

JANUARY 11, 2013 AT 3:35 AM (HTTPS://KEYHOLESOFTWARE.COM/2012/11/12/GENERATING-LARGE-EXCEL-FILES-USING-SPRING-BATCH-PART-THREE/#COMMENT-169)

Nice Article using Spring Batch with POI, but I am looking for POI generating .xls files , We have requirement of generating .xls file in our web app with huge list of Trades , like 60,000 to 1,00,000 . How will Spring Batch address that with using HSSFWorkBook instead of using SXSSFWorkbook ?

★ Loading...

Reply ↩ (https://keyholesoftware.com/2012/11/12/generating-large-excel-files-using-spring-batch-part-three/?replytocom=169#respond)

Jonny Hackett (@jhackett01) (http://twitter.com/jhackett01)

JANUARY 11, 2013 AT 10:22 AM (HTTPS://KEYHOLESOFTWARE.COM/2012/11/12/GENERATING-LARGE-EXCEL-FILES-USING-SPRING-BATCH-PART-THREE/#COMMENT-170)

Hi Rahul, thanks for the question. You can still use Spring Batch and the HSSF model together, but unfortunately Spring Batch won't address the memory issues associated with large .xls files. Because the HSSF model is based upon the older binary Excel version, it still requires you to create the entire workbook object in memory prior to writing the file out. By using the SXSSF model you don't have to keep the entire Excel workbook in memory and instead it will periodically write out portions of the file keeping the memory low. There's a small chart on the bottom of the POI Spreadsheet page that lists the different models and their features (http://poi.apache.org/spreadsheet/index.html (http://poi.apache.org/spreadsheet/index.html)). Based upon that chart, the SXSSF model is the only one that supports buffered streaming when writing files.

★ Loading...

Reply ↩ (https://keyholesoftware.com/2012/11/12/generating-large-excel-files-using-spring-batch-part-three/?replytocom=170#respond)

Pingback: Scaling Spring Batch – Step Partitioning | Keyhole Software (https://keyholesoftware.com/2013/12/09/spring-batch-partitioning/)

Sid

JUNE 16, 2014 AT 6:49 AM (HTTPS://KEYHOLESOFTWARE.COM/2012/11/12/GENERATING-LARGE-EXCEL-FILES-USING-SPRING-BATCH-PART-THREE/#COMMENT-63459)

There seems to be problem with the 2 features, i.e., sliding window of excel(SXSSFWorkbook) and commit interval of spring batch.

I tried writing 50K records. For every 10K I am creating a new sheet. So my expected output should be 10K in each.

But with SXSSFWorkbook(500) and commit-interval=500, the data seems to break in wrong way in the 2nd to last sheets.

Although the data is written correctly when its going into one sheet.

[Note: I am doing this workaround just because of the 1048575 limit of SXSSFWorkbook in excel.]

Sid.

★ Loading...

Reply ↩ (https://keyholesoftware.com/2012/11/12/generating-large-excel-files-using-spring-batch-part-three/?replytocom=63459#respond)

Jonny

JUNE 16, 2014 AT 8:28 AM (HTTPS://KEYHOLESOFTWARE.COM/2012/11/12/GENERATING-LARGE-EXCEL-FILES-USING-SPRING-BATCH-PART-THREE/#COMMENT-63507)

Thanks Sid, I'll check this out a little later in the week and see if I can replicate the issue. Are you using this exact code plus a few modifications to write the new worksheets? Or are there quite a few differences?

-jonny

★ Loading...

Reply ↩ (https://keyholesoftware.com/2012/11/12/generating-large-excel-files-using-spring-batch-part-three/?replytocom=63507#respond)

Sid

JUNE 17, 2014 AT 3:31 AM (HTTPS://KEYHOLESOFTWARE.COM/2012/11/12/GENERATING-LARGE-EXCEL-FILES-USING-SPRING-BATCH-PART-THREE/#COMMENT-64160)

Thanks Jonny for getting back. There is change from my last update.

Just a minor change in code near the creating row logic. Here is code I have changed. Besides I am not using the inner loop of 300, as my reader query already fetches around 4252362 records. Hope you can find the reason I am lossing data in the subsequent sheets. Is it bcos of some clash in sliding window of SXSSFWorkbook and commit-interval in spring batch.

```
Row row=null;
try {
row = sheet.createRow(currRow);
} catch (IllegalArgumentException iae) {
String strMessage="Invalid row number (1048576) outside allowable range (0..1048575)";
if(strMessage.equals(iae.getMessage())){
System.out.println("Exceeded limit");
currRow=0;
sheet = workbook.createSheet();
```

```
row = sheet.createRow(currRow);
}
}
```

To look at the above problem at a smaller scope. I have a query which returns only 48K records. Then I change the above logic to code below. The output(xlsx) I get is 10K records in first sheet but only 3 rows in subsequent 4 sheets.

```
if(currRow % 10000==0){
System.out.println("1 million crossed");
currRow=0;
sheet = workbook.createSheet();
}
```

In both the scenarios: commit-interval=500 and slide window for SXSSFWorkbook(100).

★ Loading...

Reply ↩ (https://keyholesoftware.com/2012/11/12/generating-large-excel-files-using-spring-batch-part-three/?replytocom=64160#respond)

Chiranjeevi

MARCH 3, 2015 AT 7:06 AM (HTTPS://KEYHOLESOFTWARE.COM/2012/11/12/GENERATING-LARGE-EXCEL-FILES-USING-SPRING-BATCH-PART-THREE/#COMMENT-358627)

Hi Jonny Hackett,

It is really an awesome blog and thanks for sharing your experience. Actually we are trying to have a custom item reader for excel sheets to load the data from excel to db. But I guess we are missing something some where while customizing. Could you please help us regarding this and provide us some sample program on this. It would be of great help.

Thanks.

★ Loading...

Reply ↩ (https://keyholesoftware.com/2012/11/12/generating-large-excel-files-using-spring-batch-part-three/?replytocom=358627#respond)

chauhanvipul87 (http://javatechknowledge.wordpress.com)

NOVEMBER 30, 2017 AT 8:17 AM (HTTPS://KEYHOLESOFTWARE.COM/2012/11/12/GENERATING-LARGE-EXCEL-FILES-USING-SPRING-BATCH-PART-THREE/#COMMENT-494928)

It is really an awesome blog and thanks for sharing your experience.

Nobody has mention about @BeforeStep & @AfterStep

★ Loading...

Reply ↩ (https://keyholesoftware.com/2012/11/12/generating-large-excel-files-using-spring-batch-part-three/?replytocom=494928#respond)

Abass

JANUARY 20, 2018 AT 2:49 AM (HTTPS://KEYHOLESOFTWARE.COM/2012/11/12/GENERATING-LARGE-EXCEL-FILES-USING-SPRING-BATCH-PART-THREE/#COMMENT-494965)

Hi Jonny, Thank you for your feedback, it is really interesting, but I wonder why not using framework like jasper report to do this kind of work besides jasper is based on apache poi. Dont you Think that it will make code base clearer and simpler? What is your opinion ?

★ Loading...

Reply ↩ (https://keyholesoftware.com/2012/11/12/generating-large-excel-files-using-spring-batch-part-three/?replytocom=494965#respond)

Ashish Sahu

MAY 25, 2018 AT 4:43 AM (HTTPS://KEYHOLESOFTWARE.COM/2012/11/12/GENERATING-LARGE-EXCEL-FILES-USING-SPRING-BATCH-PART-THREE/#COMMENT-495927)

Hi Jonny,

could be you please show me the full spring configuration file?

i am facing issue while doing the configuration of spring batch for generation of huge (millions of records).xlsx file .

your prompt reply highly appreciated

Regards,
Ashish

★ Loading...

Reply ↩ (https://keyholesoftware.com/2012/11/12/generating-large-excel-files-using-spring-batch-part-three/?replytocom=495927#respond)

marvin

SEPTEMBER 7, 2018 AT 2:33 AM (HTTPS://KEYHOLESOFTWARE.COM/2012/11/12/GENERATING-LARGE-EXCEL-FILES-USING-SPRING-BATCH-PART-THREE/#COMMENT-497712)

I am very lucky to read the article. And I appreciate that your hard work. Also I have some questions: 1.you prevent OOM by generating a big file by split into different chunks, but in the real project. the users may hope just one file. ? So . how to make it out ? Thanks again

★ Loading...

Reply ↩ (https://keyholesoftware.com/2012/11/12/generating-large-excel-files-using-spring-batch-part-three/?replytocom=497712#respond)

What Do You Think?

Enter your comment here...

Proud To Be

(https://keyholesoftware.com/company/about/microsoft-competency-partner/)

Subscribe To Dev Blog

Receive notifications of new posts by email.

Privacy Policy

Email Address

Subscribe

What We're Talking About

Keyhole Announces Kansas City Hyperledger Blockchain User Group (https://keyholesoftware.com/2018/10/29/keyhole-announces-kansas-city-hyperledger-blockchain-user-group/)

Refreshing Your Scrum (https://keyholesoftware.com/2018/10/29/refreshing-your-scrum/)

Interactive REST API Documentation with Swagger UI (https://keyholesoftware.com/2018/10/24/interactive-rest-api-documentation-with-swagger-ui/)

Release: Hyperledger Blockchain Analytics Tool (https://keyholesoftware.com/2018/10/19/release-hyperledger-blockchain-analytics-tool/)

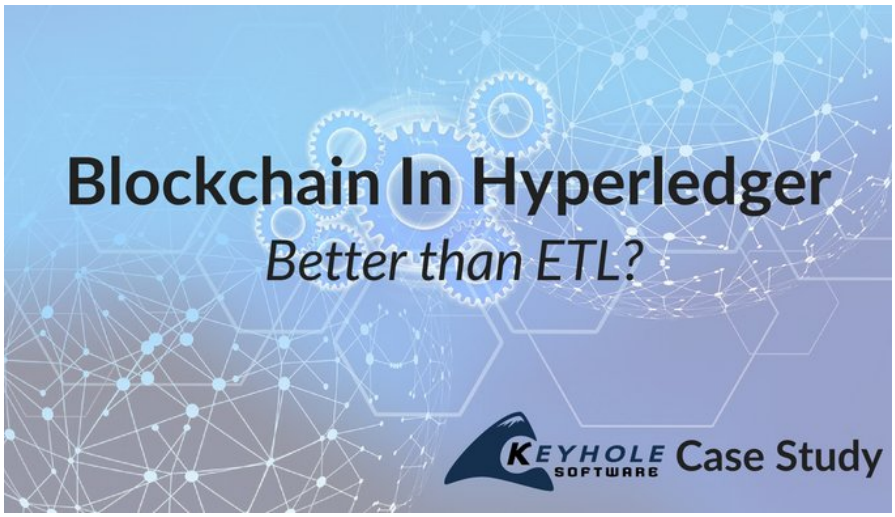
What We're Tweeting

Tweets by @KeyholeSoftware

Keyhole Software
@KeyholeSoftware

Blockchain In Hyperledger: Better Than ETL? bit.ly/2vsUr2U

[Free Case Study] Includes in-depth walkthrough of a #blockchain implemented with @Hyperledger #Fabric with a focus on the value blockchain brings to the enterprise #hyperledger #hyperledgerfabric



1h

Keyhole Software
@KeyholeSoftware

Want to learn #Microservices? Free #WhitePaper (no registration

Embed View on Twitter

HOME (HTTPS://KEYHOLESOFTWARE.COM/)
ABOUT (HTTPS://KEYHOLESOFTWARE.COM/COMPANY/ABOUT/)
SERVICES (HTTPS://KEYHOLESOFTWARE.COM/SERVICES/WHAT-WE-DO/)
BLOG (HTTPS://KEYHOLESOFTWARE.COM/BLOG/)
CONTACT (HTTPS://KEYHOLESOFTWARE.COM/CONTACT/CONTACT-AND-LOCATIONS/)
KEYHOLE TEAM LOGIN (HTTPS://SITES.GOOGLE.COM/KEYHOLESOFTWARE.COM/TEAM)

(https://facebook.com/keyholesoftware)
(https://twitter.com/keyholesoftware)
(https://www.linkedin.com/company/keyhole-software)
(https://www.youtube.com/channel/UCAIUNkXmnAPgLWnqUDpUGAQ)
(https://keyholesoftware.com/feed)

Privacy Policy