DZone

Spring Batch CSV Processing

by Michael Good MVB · Nov. 20, 17 · Java Zone · Tutorial

Welcome! Topics we will be discussing today include the essential concepts of batch processing with Spring Batch and how to import the data from a CSV into a database.

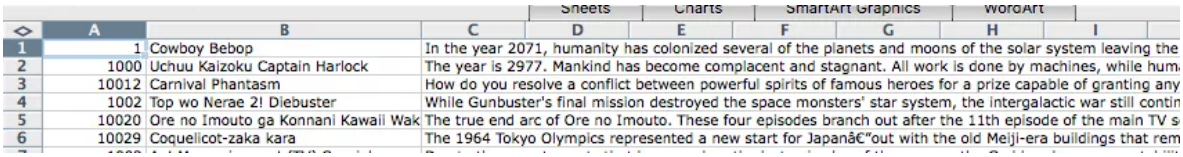
Spring Batch CSV Processing Example Application

We are building an application that demonstrates the basics of Spring Batch for processing CSV files. Our demo application will allow us to process a CSV file that contains hundreds of records of Japanese anime titles.

The CSV

I have downloaded the CSV we will be using from this GitHub repository, and it provides a pretty comprehensive list of animes.

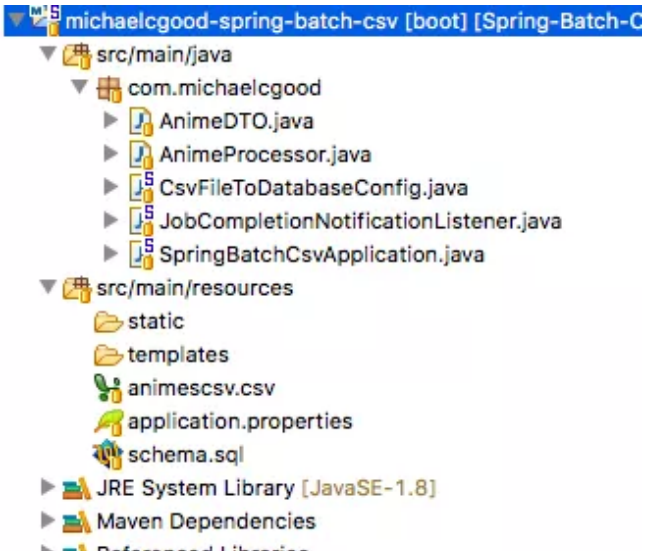
Here is a screenshot of the CSV open in Microsoft Excel

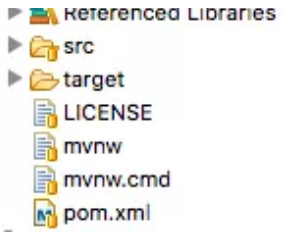


	A	B	C
1	1	Cowboy Bebop	In the year 2071, humanity has colonized several of the planets and moons of the solar system leaving the
2	1000	Uchuu Kaizoku Captain Harlock	The year is 2977. Mankind has become complacent and stagnant. All work is done by machines, while hum
3	10012	Carnival Phantasm	How do you resolve a conflict between powerful spirits of famous heroes for a prize capable of granting any
4	1002	Top wo Nerae 2! Diebuster	While Gunbuster's final mission destroyed the space monsters' star system, the intergalactic war still contin
5	10020	Ore no Imouto ga Konnani Kawaii Wak	The true end arc of Ore no Imouto. These four episodes branch out after the 11th episode of the main TV s
6	10029	Coquelicot-zaka kara	The 1964 Tokyo Olympics represented a new start for Japanâ€"out with the old Meiji-era buildings that rem

View and Download the code from GitHub.

Project Structure





Project Dependencies

Besides typical Spring Boot dependencies, we include spring-boot-starter-batch, which is the dependency for Spring Batch as the name suggests, and hsqldb for an in-memory database. We also include commons-lang3 for ToStringBuilder.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLS
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/mave
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.michaelcgood</groupId>
    <artifactId>michaelcgood-spring-batch-csv</artifactId>
    <version>0.0.1</version>
    <packaging>jar</packaging>

    <name>michaelcgood-spring-batch-csv</name>
    <description>Michael C  Good - Spring Batch CSV Example Application</description>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.5.7.RELEASE</version>
        <relativePath /> <!-- lookup parent from repository -->
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
        <java.version>1.8</java.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-batch</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
```

```
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <dependency>
            <groupId>org.hsqldb</groupId>
            <artifactId>hsqldb</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
        <!-- https://mvnrepository.com/artifact/org.apache.commons/commons-lang3 -->
        <dependency>
            <groupId>org.apache.commons</groupId>
            <artifactId>commons-lang3</artifactId>
            <version>3.6</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>


</project>
```

Model

This is a POJO that models the fields of an anime. The fields are:

- ID. For the sake of simplicity, we treat the ID as a String. However, this could be changed to another data type such as an Integer or Long.
- Title. This is the title of the anime and it is appropriate for it to be a String.
- Description. This is the description of the anime, which is longer than the title, and it can also be treated as a String.

What is important to note is our class constructor for the three fields: public AnimeDTO(String id, String title, String description). This will be used in our application. Also, as usual, we need to make a default constructor with no parameters or else Java will throw an error.

```
package com.michaelcgood;

import org.apache.commons.lang3.builder.ToStringBuilder;
/**
 * Contains the information of a single anime
 *
 * @author Michael C Good michaelcgood.com
 */

public class AnimeDTO {

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    private String id;

    private String title;
    private String description;

    public AnimeDTO(){

    }
```

```

    public AnimeDTO(String id, String title, String description){
        this.id = id;
        this.title = title;
        this.description = title;
    }

       @Override
        public String toString() {
           return new ToStringBuilder(this)
                   .append("id", this.id)
                   .append("title", this.title)
                   .append("description", this.description)
                   .toString();
       }

}
```

CSV File to Database Configuration

There is a lot going on in this class and it is not all written at once, so we are going to go through the code in steps. Visit GitHub to see the code in its entirety.

Reader

As the Spring Batch documentation states FlatFileIteamReader will “read lines of data from a flat file that typically describe records with fields of data defined by fixed positions in the file or delimited by some special character (e.g. Comma)”.

We are dealing with a CSV, so of course the data is delimited by a comma, making this the perfect for use with our file.

```
@Bean
public FlatFileItemReader < AnimeDTO > csvAnimeReader() {
    FlatFileItemReader < AnimeDTO > reader = new FlatFileItemReader < AnimeDTO > ();
    reader.setResource(new ClassPathResource("animescsv.csv"));
    reader.setLineMapper(new DefaultLineMapper < AnimeDTO > () {
        {
            setLineTokenizer(new DelimitedLineTokenizer() {
                {
                    setNames(new String[] {
                        "id",
                        "title",
                        "description"
                    });
                }
            });
```

```
            setFieldSetMapper(new BeanWrapperFieldSetMapper < AnimeDTO > () {
                {
                    setTargetType(AnimeDTO.class);
                }
            });
        }
    });
    return reader;
}
```

Important points:

- FlatFileItemReader is parameterized with a model. In our case, this is AnimeDTO.
- FlatFileItemReader must set a resource. It uses *setResource* method. Here we set the resource to *animescsv.csv*
- *setLineMapper* method converts Strings to objects representing the item. Our String will be an anime record consisting of an id, title, and description. This String is made into an object. Note that *DefaultLineMapper* is parameterized with our model, AnimeDTO.
- However, LineMapper is given a raw line, which means there is work that needs to be done to map the fields appropriately. The line must be tokenized into a FieldSet, which *DelimitedLineTokenizer* takes care of. DelimitedLineTokenizer returns a FieldSet.
- Now that we have a FieldSet, we need to map it. *setFieldSetMapper* is used for taking the FieldSet object and mapping its contents to a DTO, which is AnimeDTO in our case.

Processor

If we want to transform the data before writing it to the database, an ItemProcessor is necessary. Our code does not actually apply any business logic to transform the data, but we allow for the capability to.

Processor in CSVFILETODATABASECONFIG.java

csvAnimeProcessor returns a new instance of the AnimeProcessor object which we review below.

```
@Bean
ItemProcessor<AnimeDTO, AnimeDTO> csvAnimeProcessor() {
    return new AnimeProcessor();
}
```

ANIMEPROCESSOR.java

If we wanted to apply business logic before writing to the database, you could manipulate the Strings

before writing to the database. For instance, you could add *toUpperCase()* after *getTitle* to make the title upper case before writing to the database. However, I decided not to do that or apply any other business logic for this example processor, so no manipulation is being done. The Processor is here simply for demonstration.

```
package com.michaelcgood;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import org.springframework.batch.item.ItemProcessor;

public class AnimeProcessor implements ItemProcessor<AnimeDTO, AnimeDTO> {

    private static final Logger log = LoggerFactory.getLogger(AnimeProcessor.class);

    @Override
    public AnimeDTO process(final AnimeDTO AnimeDTO) throws Exception {

        final String id = AnimeDTO.getId();
        final String title = AnimeDTO.getTitle();
        final String description = AnimeDTO.getDescription();

        final AnimeDTO transformedAnimeDTO = new AnimeDTO(id, title, description);

        log.info("Converting (" + AnimeDTO + ") into (" + transformedAnimeDTO + ")");

        return transformedAnimeDTO;
    }

}
```

Writer

The *csvAnimeWriter* method is responsible for actually writing the values into our database. Our database is an in-memory HSQLDB, however, this application allows us to easily swap out one database for another. The *dataSource* is autowired.

```
@Bean
public JdbcBatchItemWriter<AnimeDTO> csvAnimeWriter() {
    JdbcBatchItemWriter<AnimeDTO> excelAnimeWriter = new JdbcBatchItemWriter<AnimeDTO>();

    excelAnimeWriter.setItemSqlParameterSourceProvider(new BeanPropertyItemSqlParameterSo

    excelAnimeWriter.setSql("INSERT INTO animes (id, title, description) VALUES (:id, :ti

    excelAnimeWriter.setDataSource(dataSource);
        return excelAnimeWriter;
```

```
        return excelAnimeWriter;
}
```

Step

A Step is a domain object that contains an independent, sequential phase of a batch job and contains all of the information needed to define and control the actual batch processing.

Now that we've created the reader and processor for data we need to write it. For the reading, we've been using chunk-oriented processing, meaning we've been reading the data one at a time. Chunk-oriented processing also includes creating 'chunks' that will be written out, within a transaction boundary. For chunk-oriented processing, you set a commit interval and once the number of items read equals the commit interval that has been set, the entire chunk is written out via the ItemWriter, and the transaction is committed. We set the chunk interval size to 1.

I suggest reading the Spring Batch documentation about chunk-oriented processing.

Then the reader, processor, and writer call the methods we wrote.

```
@Bean
public Step csvFileToDatabaseStep() {
    return stepBuilderFactory.get("csvFileToDatabaseStep")
            .<AnimeDTO, AnimeDTO>chunk(1)
            .reader(csvAnimeReader())
            .processor(csvAnimeProcessor())
            .writer(csvAnimeWriter())
            .build();
}
```

Job

A Job consists of Steps. We pass a parameter into the Job below because we want to track the completion of the Job.

```
@Bean
Job csvFileToDatabaseJob(JobCompletionNotificationListener listener) {
    return jobBuilderFactory.get("csvFileToDatabaseJob")
            .incrementer(new RunIdIncrementer())
            .listener(listener)
            .flow(csvFileToDatabaseStep())
            .end()
            .build();
}
```

Job Completion Notification Listener

The class below autowires the JdbcTemplate because we've already set the dataSource and we want to easily make our query. The results of our are query are a list of AnimeDTO objects. For each object returned, we will create a message in our console to show that the item has been written to the database.

SQL

We need to create a schema for our database. As mentioned, we have made all fields Strings for ease of use, so we have made their data types VARCHAR.

Main

This is a standard class with main(). As the Spring Documentation states, *@SpringBootApplication* is a convenience annotation that includes *@Configuration*, *@EnableAutoConfiguration*, *@EnableWebMvc*, and *@ComponentScan*.

Demo

Converting

The FieldSet is fed through the processor and "Converting" is printed to the console.

```
TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
JobLauncherCommandLineRunner   : Running default command line with: [--spring.output.ansi.enabled=always]
support.SimpleJobLauncher      : Job: [FlowJob: [name=csvFileToDatabaseJob]] launched with the following parameters: [{run.id=1, -spring.output.ansi.enabled=al
core.job.SimpleStepHandler     : Executing step: [csvFileToDatabaseStep]
lcgood.AnimeProcessor          : Converting (com.michaelcgood.AnimeDTO@55877274[id=1,title=Cowboy Bebop,description=In the year 2071, humanity has colonized se
lcgood.AnimeProcessor          : Converting (com.michaelcgood.AnimeDTO@3a4a5f3c[id=1000,title=Uchuu Kaizoku Captain Harlock,description=The year is 2977. Mankir
lcgood.AnimeProcessor          : Converting (com.michaelcgood.AnimeDTO@19b5214b[id=10012,title=Carnival Phantasm,description=How do you resolve a conflict betw
lcgood.AnimeProcessor          : Converting (com.michaelcgood.AnimeDTO@3820cfe[id=1002,title=Top wo Nerae 2! Diebuster,description=While Gunbuster's final missi
lcgood.AnimeProcessor          : Converting (com.michaelcgood.AnimeDTO@6b8b5020[id=10020,title=Ore no Imouto ga Konnani Kawaii Wake ga Nai Specials,description=
lcgood.AnimeProcessor          : Converting (com.michaelcgood.AnimeDTO@5eb87338[id=10029,title=Coquelicot-zaka kara,description=The 1964 Tokyo Olympics represe
lcgood.AnimeProcessor          : Converting (com.michaelcgood.AnimeDTO@1785d194[id=1003,title=Aa! Megami-sama! (TV) Specials,description=Due to the recent even
lcgood.AnimeProcessor          : Converting (com.michaelcgood.AnimeDTO@1b8aaeab[id=10030,title=Bakuman. 2,description=Mashiro and Takagi start working with a n
lcgood.AnimeProcessor          : Converting (com.michaelcgood.AnimeDTO@d87d449[id=10033,title=Toriko,description=Welcome to the Gourmet Age, where happiness is
lcgood.AnimeProcessor          : Converting (com.michaelcgood.AnimeDTO@686cf8ad[id=1004,title=Kanojo to Kanojo no Neko,description=Do you believe in love at fi
```

Discovering New Items In Database

When the Spring Batch Job is finished, we select all the records and print them out to the console individually.

```
aelcgood.AnimeProcessor          : Converting (com.michaelcgood.AnimeDTO@5a08d301[id=9989,title=Ano Hi Mita Hana no Namae wo Bokutachi wa Mada Shiranai.,
aelcgood.AnimeProcessor          : Converting (com.michaelcgood.AnimeDTO@3f919230[id=9996,title=Hyouge Mono,description=The story is set during Japan's
ompletionNotificationListener    : ============ JOB FINISHED ============ Verifying the results....

ompletionNotificationListener    : Discovered <com.michaelcgood.AnimeDTO@eda7dd3[id=1,title=Cowboy Bebop,description=Cowboy Bebop]> in the database.
ompletionNotificationListener    : Discovered <com.michaelcgood.AnimeDTO@9d7a853[id=1000,title=Uchuu Kaizoku Captain Harlock,description=Uchuu Kaizoku C
ompletionNotificationListener    : Discovered <com.michaelcgood.AnimeDTO@71c69628[id=10012,title=Carnival Phantasm,description=Carnival Phantasm]> in th
ompletionNotificationListener    : Discovered <com.michaelcgood.AnimeDTO@4dc16fc9[id=1002,title=Top wo Nerae 2! Diebuster,description=Top wo Nerae 2! Di
ompletionNotificationListener    : Discovered <com.michaelcgood.AnimeDTO@1028a747[id=10020,title=Ore no Imouto ga Konnani Kawaii Wake ga Nai Specials,de
ompletionNotificationListener    : Discovered <com.michaelcgood.AnimeDTO@24890021[id=10029,title=Coquelicot-zaka kara,description=Coquelicot-zaka kara]>
ompletionNotificationListener    : Discovered <com.michaelcgood.AnimeDTO@613ba54e[id=1003,title=Aa! Megami-sama! (TV) Specials,description=Aa! Megami-sa
```

Batch Process Complete

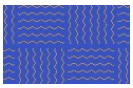
When the Batch Process is complete this is what is printed to the console.

Conclusion

Spring Batch builds upon the POJO-based development approach and user-friendliness of the Spring Framework's to make it easy for developers to create enterprise-gradee batch processing.

The source code is on GitHub.

Like This Article? Read More From DZone



Pseudo-Static Row Mappers, a Healthy Alternative to Static Row Mapping

An Introduction to Spring Batch

Who Needs Batch These Days?

Free DZone Refcard
Java Containerization

Topics: JAVA , CSV , BATCH PROCESSING , SPRING BATCH , TUTORIAL

Published at DZone with permission of Michael Good , DZone MVB. See the original article here.
Opinions expressed by DZone contributors are their own.