Spring Batch

An Introduction

Duncan McIntyre

Tanzu Solutions Engineer at VMware

June 2020

Agenda

What Is Batch Processing?

Why Modernize

Why Do We Need A Framework?

What Is Spring Batch?

Spring Task – A Lightweight Alternative

When To Use Batch

Events Dear Boy, Events

Scaling

Writing A Spring Batch Job

What Is Batch Processing?

The Lifecycle Of A Batch Job

Batch vs Non-Batch

Lifecycle, State

Batch Processing

Automated processing of large volumes of data without user interaction. Typically time-based - such as month-end calculations, notices, or correspondence.

Periodic application of complex business rules processed repetitively across very large data sets (for example, insurance benefit determination or rate adjustments).

Integration of information from internal and external systems requiring formatting, validation, and processing in a transactional manner into the system of record. Used to process billions of transactions every day for enterprises.

Non-Batch Processing

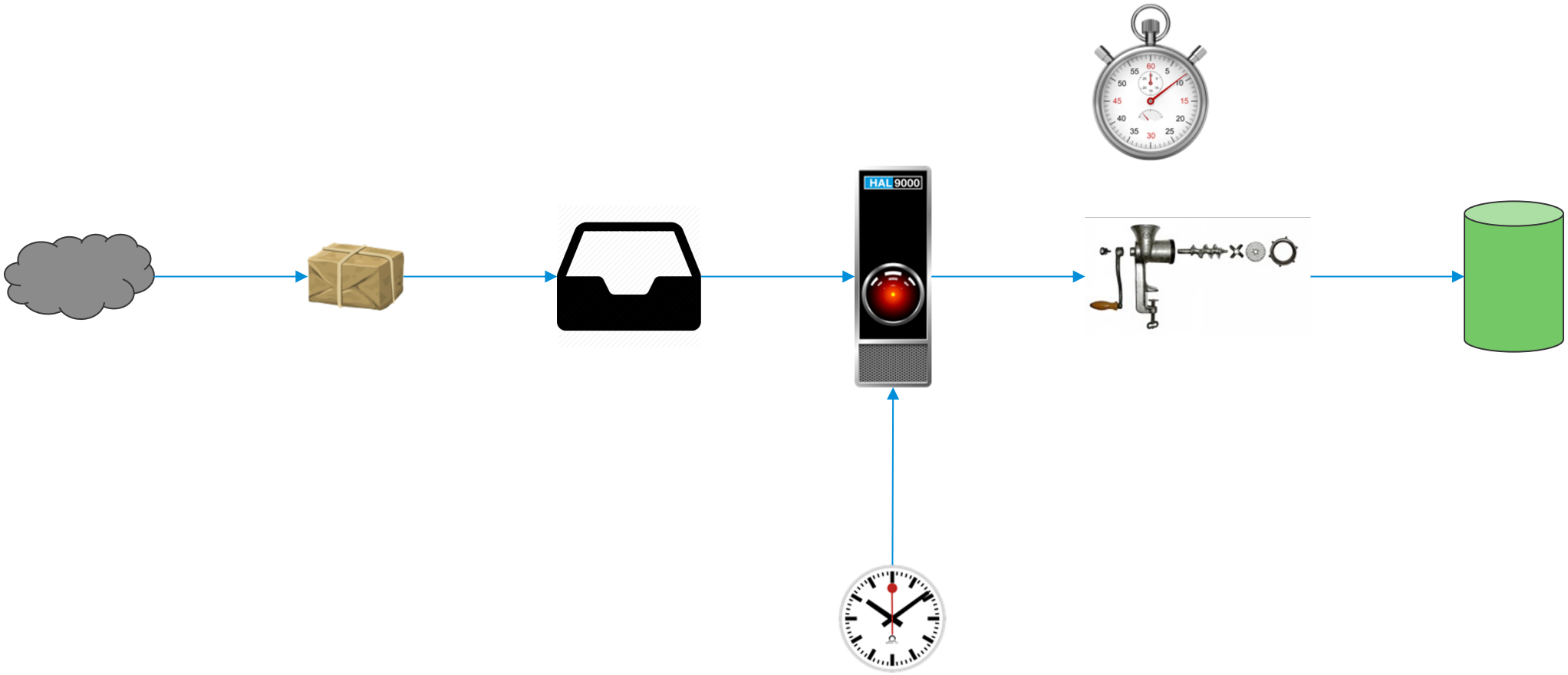
Traditional applications which run for an extended period (for ever if possible!) interacting either with end-users or with other systems.

May be stateless, or may have complex state which is not easily captured.

If the application dies, there may be no concept of “picking up where it left off”.

Lifecycle Of A Batch Job - Example

Not To Scale



Why Modernize?

Our existing batch works fine

Reasons To Modernize A Batch Application

And Reasons To Leave It Alone

Modernize!

Job is written in a legacy language which nobody uses any more

Job can't be scaled any further

Batch is no longer appropriate – customers expect updates in real-time (Events!)

Cost savings

Easier and faster to develop and test in a cloud environment**

Cloud-based architecture could be deployed to multiple infrastructures**

Leave It Alone!

Job works fine and meets all NFRs

Job needs to be close to the data/mainframe for performance reasons

Can't take advantage of the cloud because of data gravity

Why Do We Need A Framework?

Standards Are Good

Bad Things Happen

Knowledge Is Power

Why Do We Need A Framework?

Just Use Duncan's Code ☺

Bad Things Happen

Jobs break

Jobs run slowly

Knowledge Is Power

How far did the job get?

Can the job be restarted?

Do broken jobs cleanup after themselves?

Can we scale up bits of the job to make it go faster?

Spring Batch

Why Use Spring Batch?

If It Didn't Exist, You'd Have To Invent It

Spring Batch developed by SpringSource (VMware) and Accenture to capture and codify decades of shared experience writing and running batch jobs in multiple languages and for multiple systems.

Spring Batch (2008) was the inspiration for JBatch - JSR-352 (2014), and is a conformant implementation.

Fully integrated with the rest of the Spring ecosystem

- consume Spring Integration components
- deploy using Spring Cloud Data Flow
- wrap multiple jobs using Spring Cloud Task

Battle tested

What Is Spring Batch?

It Is

A framework built on Spring for writing batch processing applications in Java

A set of architectural patterns which provide a standard way to implement desirable features such as observability, restartability and scalability

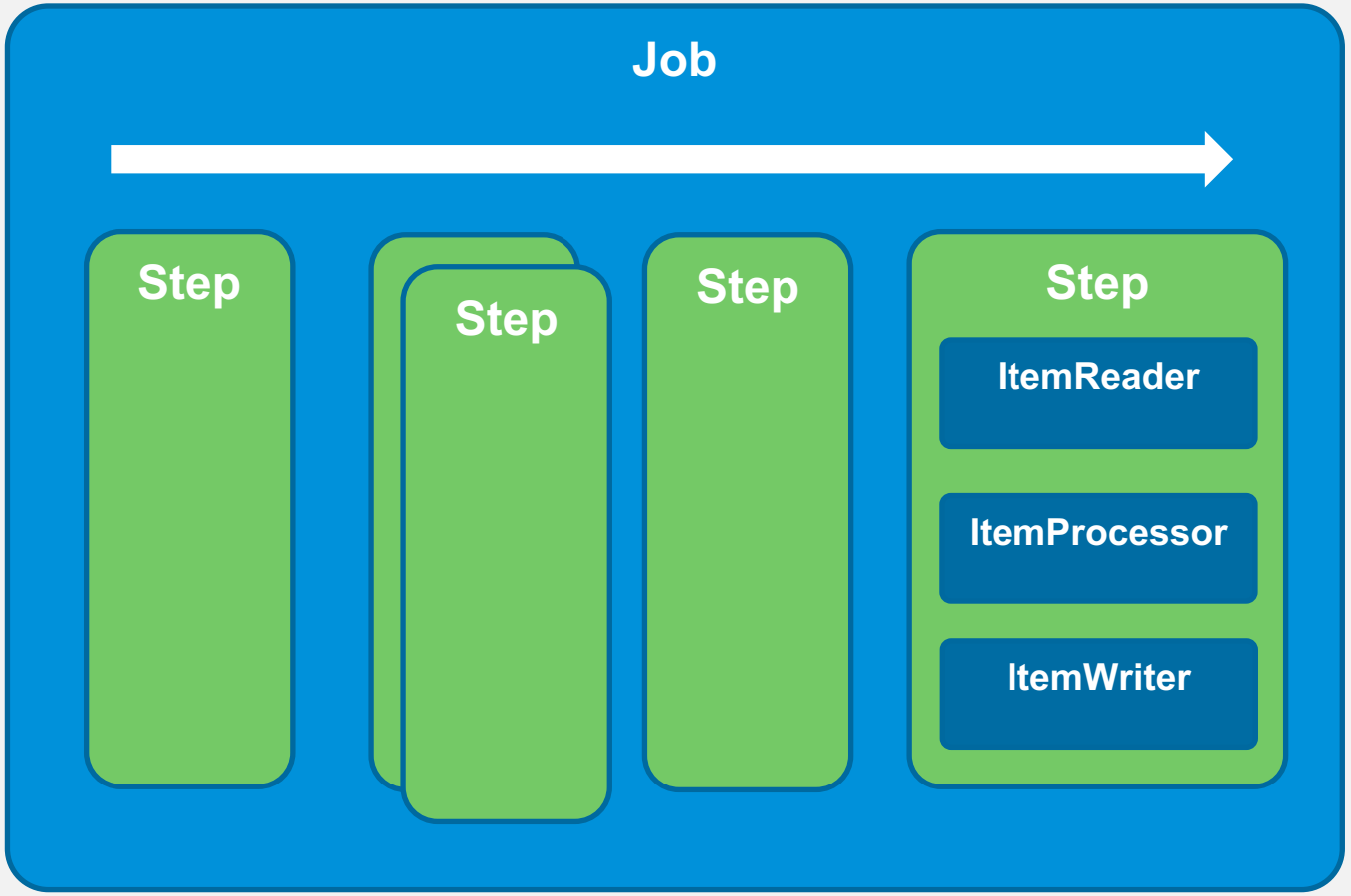
It Is Not

A scheduler

Image: http://www.cullysbakery.com/breads

(Some) Spring Batch Components

Job, Step, Reader, Processor, Writer



A Job is simply a container for Step instances. It combines multiple steps that belong logically together in a flow and allows for configuration of properties global to all steps, such as restartability

The job configuration contains:

- The simple name of the job.
- Definition and ordering of Step instances
- Whether or not the job is restartable

https://docs.spring.io/spring-batch/docs/current/reference/html/domain.html#job

Spring Task – A Lighter Alternative To Spring Batch

Image: https://www.heraldscotland.com/life_style/13084760.simply-special-ode-to-the-humble-oatcake/

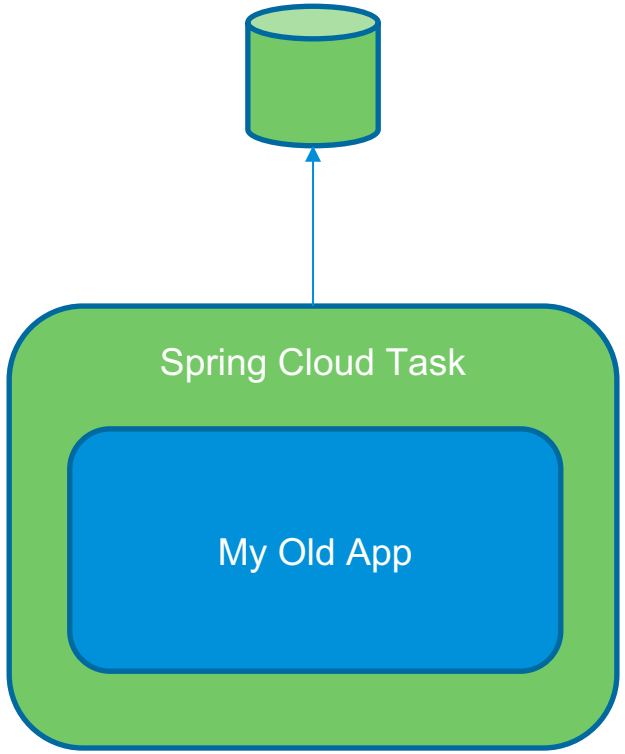
Spring Cloud Task

Keeping Track Of Tasks

Spring Cloud Task

Wraps a Spring Boot Application

Records the app's executions and exit status



Field	Description
executionid	The unique ID for the task's run.
exitCode	The exit code generated from an ExitCodeExceptionMapper implementation. If there is no exit code generated but an ApplicationFailedEvent is thrown, 1 is set. Otherwise, it is assumed to be 0.
taskName	The name for the task, as determined by the configured TaskNameResolver.
startTime	The time the task was started, as indicated by the SmartLifecycle#start call.
endTime	The time the task was completed, as indicated by the ApplicationReadyEvent.
exitMessage	Any information available at the time of exit. This can programmatically be set by a TaskExecutionListener.
errorMessage	If an exception is the cause of the end of the task (as indicated by an ApplicationFailedEvent), the stack trace for that exception is stored here.
arguments	A List of the string command line arguments as they were passed into the executable boot application.

When To Use Spring Batch, And How To Start

Task v. Batch

Which One To Use

Batch

Keep track of Step execution so a job can be be stopped and restarted

Implement interfaces to allow horizontal scaling

Listen for events in the lifecycle of a Job and react appropriately

Task

Existing code works fine, just need a consistent way of keeping track of executions

Getting Started With Batch

Basic Data Transformation Using Existing ItemReader and ItemWriter

Here's one I prepared earlier ->

Item Reader	Description
AbstractItemCountingItemStreamItemReader	Abstract base class that provides basic restart capabilities by counting the number of items returned from an ItemReader.
AggregateItemReader	An ItemReader that delivers a list as its item, storing up objects from the injected ItemReader until they are ready to be packed out as a collection. This class must be used as a wrapper for a custom ItemReader that can identify the record boundaries. The custom reader should mark the beginning and end of records by returning an AggregateItem which responds true to its query methods isHeader() and isFooter(). Note that this reader is not part of the library of readers provided by Spring Batch but given as a sample in spring-batch-samples.
AmqpItemReader	Given a Spring AmqpTemplate, it provides synchronous receive methods. The receiveAndConvert() method lets you receive POJO objects.
KafkaItemReader	An ItemReader that reads messages from an Apache Kafka topic. It can be configured to read messages from multiple partitions of the same topic. This reader stores message offsets in the execution context to support restart capabilities.
FlatFileItemReader	Reads from a flat file. Includes ItemStream and Skippable functionality. See FlatFileItemReader.
HibernateCursorItemReader	Reads from a cursor based on an HQL query. See Cursor-based ItemReaders.
HibernatePagingItemReader	Reads from a paginated HQL query
ItemReaderAdapter	Adapts any class to the ItemReader interface.
JdbcCursorItemReader	Reads from a database cursor via JDBC. See Cursor-based ItemReaders.
JdbcPagingItemReader	Given an SQL statement, pages through the rows, such that large datasets can be read without running out of memory.
JmsItemReader	Given a Spring JmsOperations object and a JMS Destination or destination name to which to send errors, provides items received through the injected JmsOperations#receive() method.
JpaPagingItemReader	Given a JPQL statement, pages through the rows, such that large datasets can be read without running out of memory.
ListItemReader	Provides the items from a list, one at a time.
MongoItemReader	Given a MongoOperations object and a JSON-based MongoDB query, provides items received from the MongoOperations#find() method.
Neo4jItemReader	Given a Neo4jOperations object and the components of a Cyhper query, items are returned as the result of the Neo4jOperations.query method.
RepositoryItemReader	Given a Spring Data PagingAndSortingRepository object, a Sort, and the name of method to execute, returns items provided by the Spring Data repository implementation.
StoredProcedureItemReader	Reads from a database cursor resulting from the execution of a database stored procedure. See StoredProcedureItemReader
StaxEventItemReader	Reads via StAX. see StaxEventItemReader.
JsonItemReader	Reads items from a Json document. see JsonItemReader.

Item Writer	Description
AbstractItemStreamItemWriter	Abstract base class that combines the ItemStream and ItemWriter interfaces.
AmqpItemWriter	Given a Spring AmqpTemplate, it provides for a synchronous send method. The convertAndSend(Object) method lets you send POJO objects.
CompositeItemWriter	Passes an item to the write method of each in an injected List of ItemWriter objects.
FlatFileItemWriter	Writes to a flat file. Includes ItemStream and Skippable functionality. See FlatFileItemWriter.
GemfireItemWriter	Using a GemfireOperations object, items are either written or removed from the Gemfire instance based on the configuration of the delete flag.
HibernateItemWriter	This item writer is Hibernate-session aware and handles some transaction-related work that a non-"hibernate-aware" item writer would not need to know about and then delegates to another item writer to do the actual writing.
ItemWriterAdapter	Adapts any class to the ItemWriter interface.
JdbcBatchItemWriter	Uses batching features from a PreparedStatement, if available, and can take rudimentary steps to locate a failure during a flush.
JmsItemWriter	Using a JmsOperations object, items are written to the default queue through the JmsOperations#convertAndSend() method.
JpaItemWriter	This item writer is JPA EntityManager-aware and handles some transaction-related work that a non-"JPA-aware" ItemWriter would not need to know about and then delegates to another writer to do the actual writing.
KafkaItemWriter	Using a KafkaTemplate object, items are written to the default topic through the KafkaTemplate#sendDefault(Object, Object) method using a Converter to map the key from the item. A delete flag can also be configured to send delete events to the topic.
MimeMessageItemWriter	Using Spring's JavaMailSender, items of type MimeMessage are sent as mail messages.
MongoItemWriter	Given a MongoOperations object, items are written through the MongoOperations.save(Object) method. The actual write is delayed until the last possible moment before the transaction commits.
Neo4jItemWriter	Given a Neo4jOperations object, items are persisted through the save(Object) method or deleted through the delete(Object) per the ItemWriter's configuration
PropertyExtractingDelegatingItemWriter	Extends AbstractMethodInvokingDelegator creating arguments on the fly. Arguments are created by retrieving the values from the fields in the item to be processed (through a SpringBeanWrapper), based on an injected array of field names.
RepositoryItemWriter	Given a Spring Data CrudRepository implementation, items are saved through the method specified in the configuration.
StaxEventItemWriter	Uses a Marshaller implementation to convert each item to XML and then writes it to an XML file using StAX.
JsonFileItemWriter	Uses a JsonObjectMarshaller implementation to convert each item to Json and then writes it to an Json file.

that's enough food-based jokes - Ed

Isn’t Everything Event-Driven These Days?

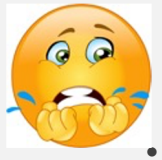
Why bother with batch at all?

Batch vs Events

Why Use One Or The Other?

Batch

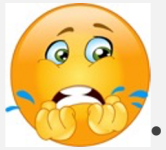
- Only uses resources intermittently
- Can maximize usage of a running resource (scale out after scale-up)
- Easy to understand and test
- Deterministic (?)



- Needs scheduling
- Scaling needs to be explicitly coded for

Events

- No scheduling requirement
- Handles scale by design

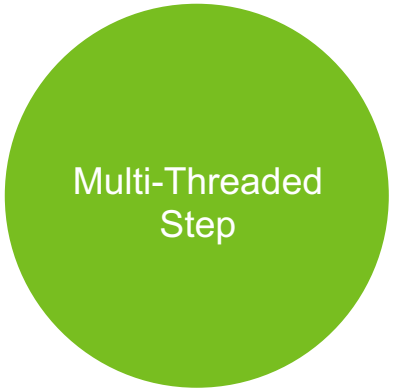


- Uses resources constantly, maybe inefficiently (functions mitigate this)
- Existing systems may not be event sources

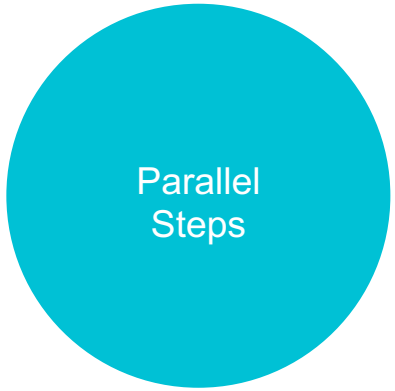
Scaling Up

Parallel Processing Strategies

Local and Remote



Run any individual Step in multiple threads on one host



Run two or more Steps in parallel on one host

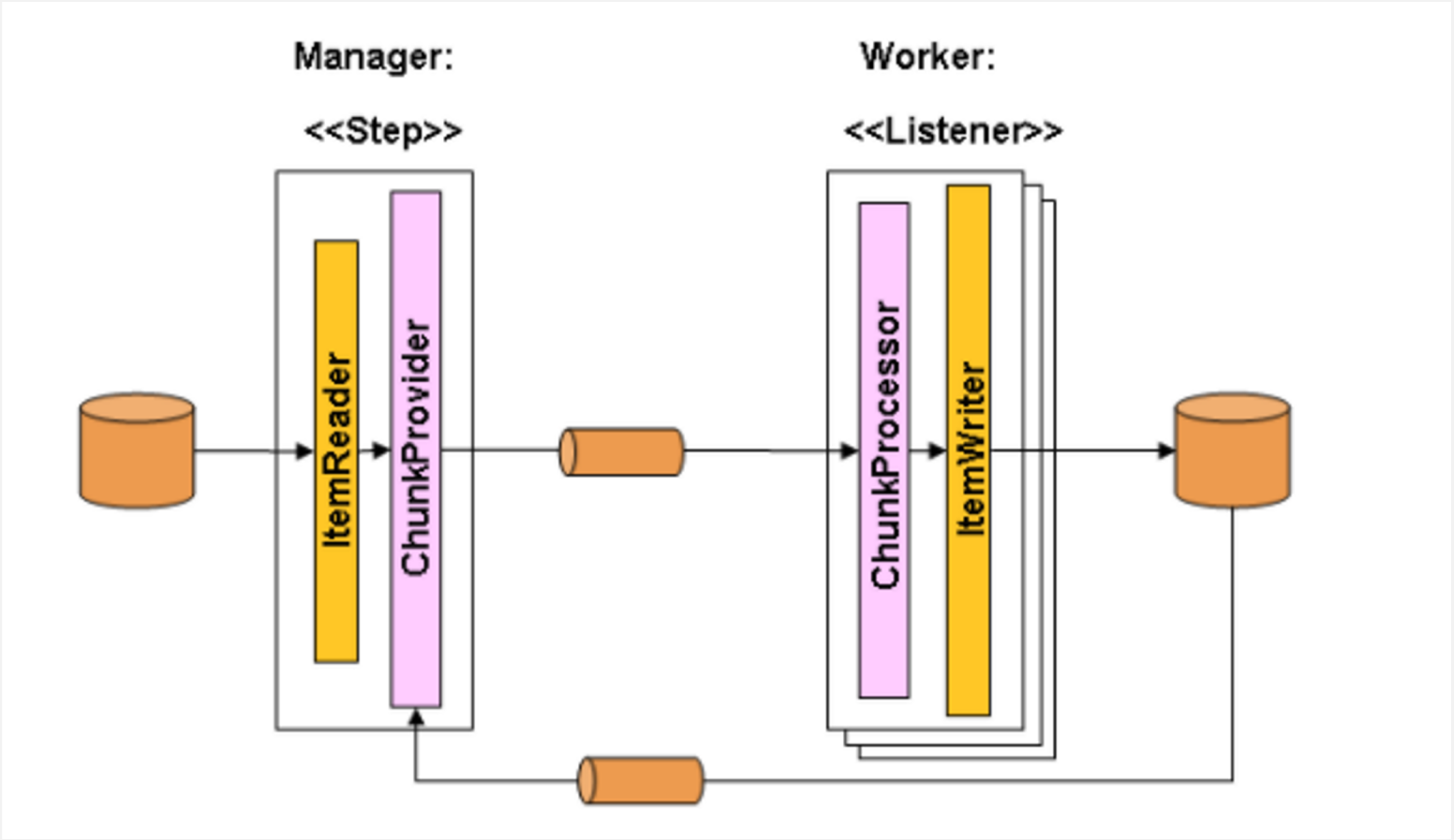
Manager Step slices the input data into Chunks and posts them to middleware (message queue).

Worker Steps take chunks from the queue for processing.

Like Remote Chunking, but the Manager Step knows how to talk to the compute fabric to directly schedule remote Step execution

Remote Chunking

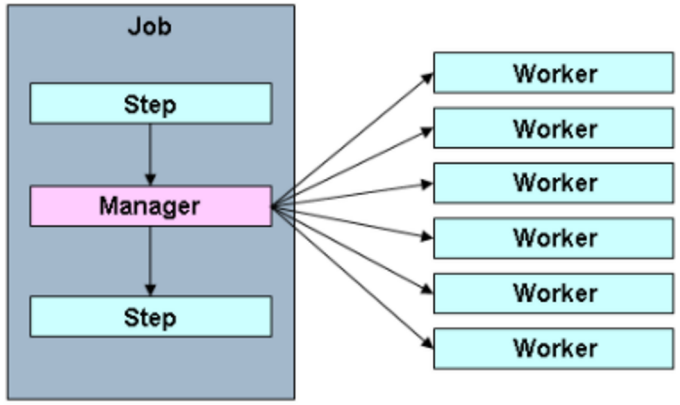
Sharing The Load With Messaging

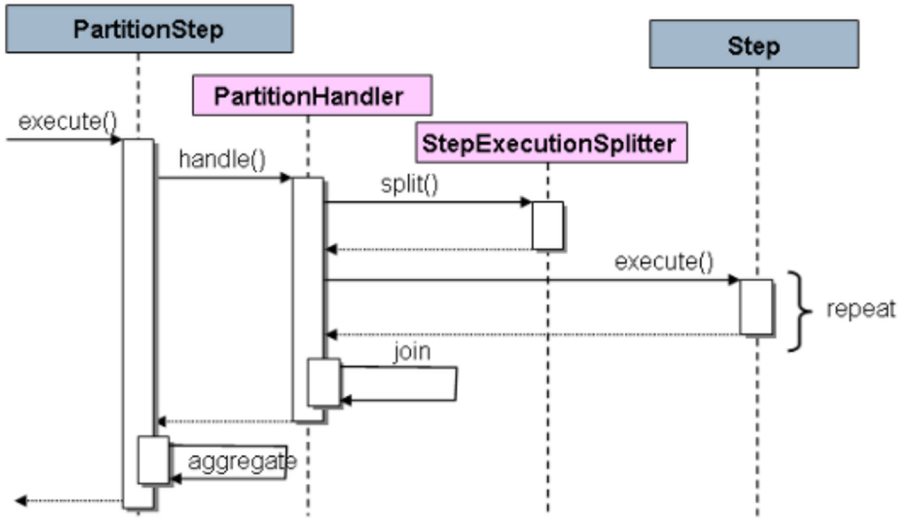


Middleware must guarantee to not lose messages

Partitioning

Cloud Native Processing





PartitionHandler knows how to get a remote Step scheduled and executed on a fabric.

- Cloud Foundry
- Kubernetes
- RMI
- EJB
- …

Let’s Write A Job

https://spring.io/guides/gs/batch-processing/

Create A Business Class

A Simple POJO

We'll write a batch job to transform people's names

```
package com.example.batchprocessing;

public class Person {

  private String lastName;
  private String firstName;

  public Person() {
  }

  public Person(String firstName, String lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }

  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }

  public String getFirstName() {
    return firstName;
  }

  public String getLastName() {
    return lastName;
  }

  public void setLastName(String lastName) {
    this.lastName = lastName;
  }

  @Override
  public String toString() {
    return "firstName: " + firstName + ", lastName: " + lastName;
  }

}
```

COPY

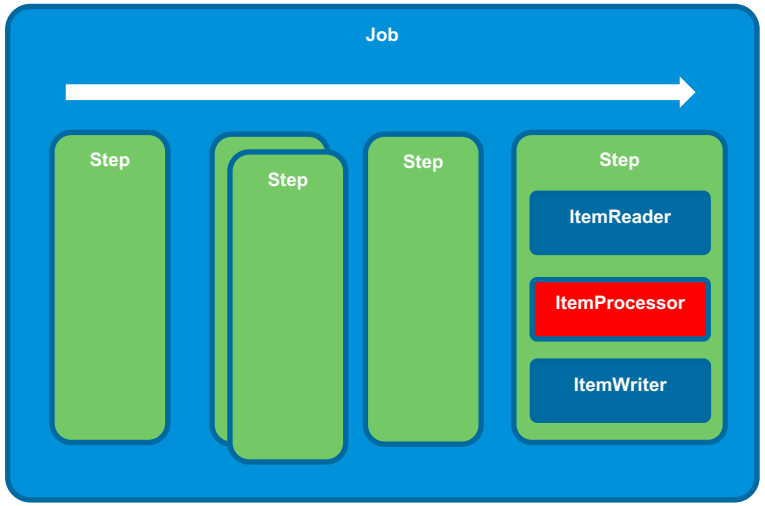
The Business Logic

Writing An ItemProcessor

```java
package com.example.batchprocessing;

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

import org.springframework.batch.item.ItemProcessor;

public class PersonItemProcessor implements ItemProcessor<Person, Person> {

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

  @Override
  public Person process(final Person person) throws Exception {
    final String firstName = person.getFirstName().toUpperCase();
    final String lastName = person.getLastName().toUpperCase();

    final Person transformedPerson = new Person(firstName, lastName);

    log.info("Converting (" + person + ") into (" + transformedPerson + ")");

    return transformedPerson;
  }

}
```

COPY

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

  O process(I item) throws Exception;

}
```



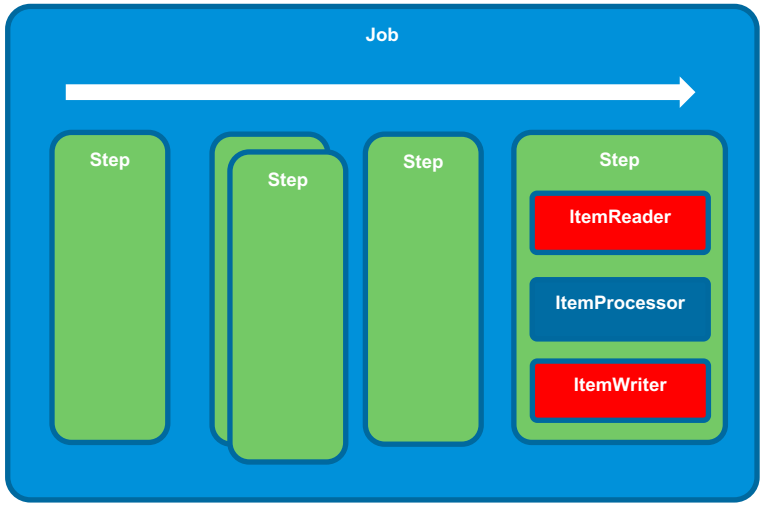
ItemReader and ItemWriter

Moving The Data

```
@Bean
public FlatFileItemReader<Person> reader() {
  return new FlatFileItemReaderBuilder<Person>()
    .name("personItemReader")
    .resource(new ClassPathResource("sample-data.csv"))
    .delimited()
    .names(new String[]{"firstName", "lastName"})
    .fieldSetMapper(new BeanWrapperFieldSetMapper<Person>() {{
      setTargetType(Person.class);
    }})
    .build();
}

@Bean
public PersonItemProcessor processor() {
  return new PersonItemProcessor();
}

@Bean
public JdbcBatchItemWriter<Person> writer(DataSource dataSource) {
  return new JdbcBatchItemWriterBuilder<Person>()
    .itemSqlParameterSourceProvider(new
BeanPropertyItemSqlParameterSourceProvider<>())
    .sql("INSERT INTO people (first_name, last_name) VALUES (:firstName,
:lastName)")
    .dataSource(dataSource)
    .build();
}
```

COPY

```
public interface ItemReader<T> {

  T read() throws Exception...;

}

public interface ItemWriter<T> {

  void write(List<? extends T> items) throws
Exception...;

}
```



Wiring Things Up

Enabling Jobs Using Annotations

Annotations enable a set of supporting beans, including an in-memory job respository

```
@Configuration
@EnableBatchProcessing
public class BatchConfiguration {

  @Autowired
  public JobBuilderFactory jobBuilderFactory;

  @Autowired
  public StepBuilderFactory stepBuilderFactory;

    ...

}
```

COPY

Wiring Things Up

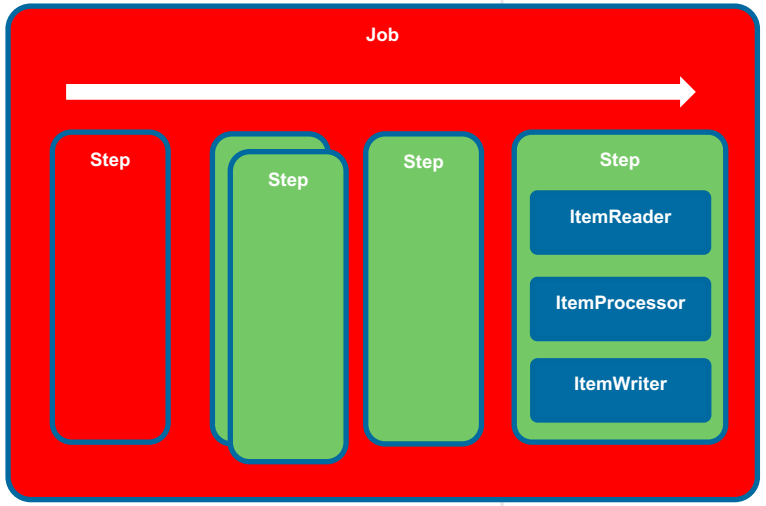
Defining The Job and Step Using Builders

Builders are used to declare the sequence of Steps in a Job and other configuration.

Builders can also be used to define a Step.

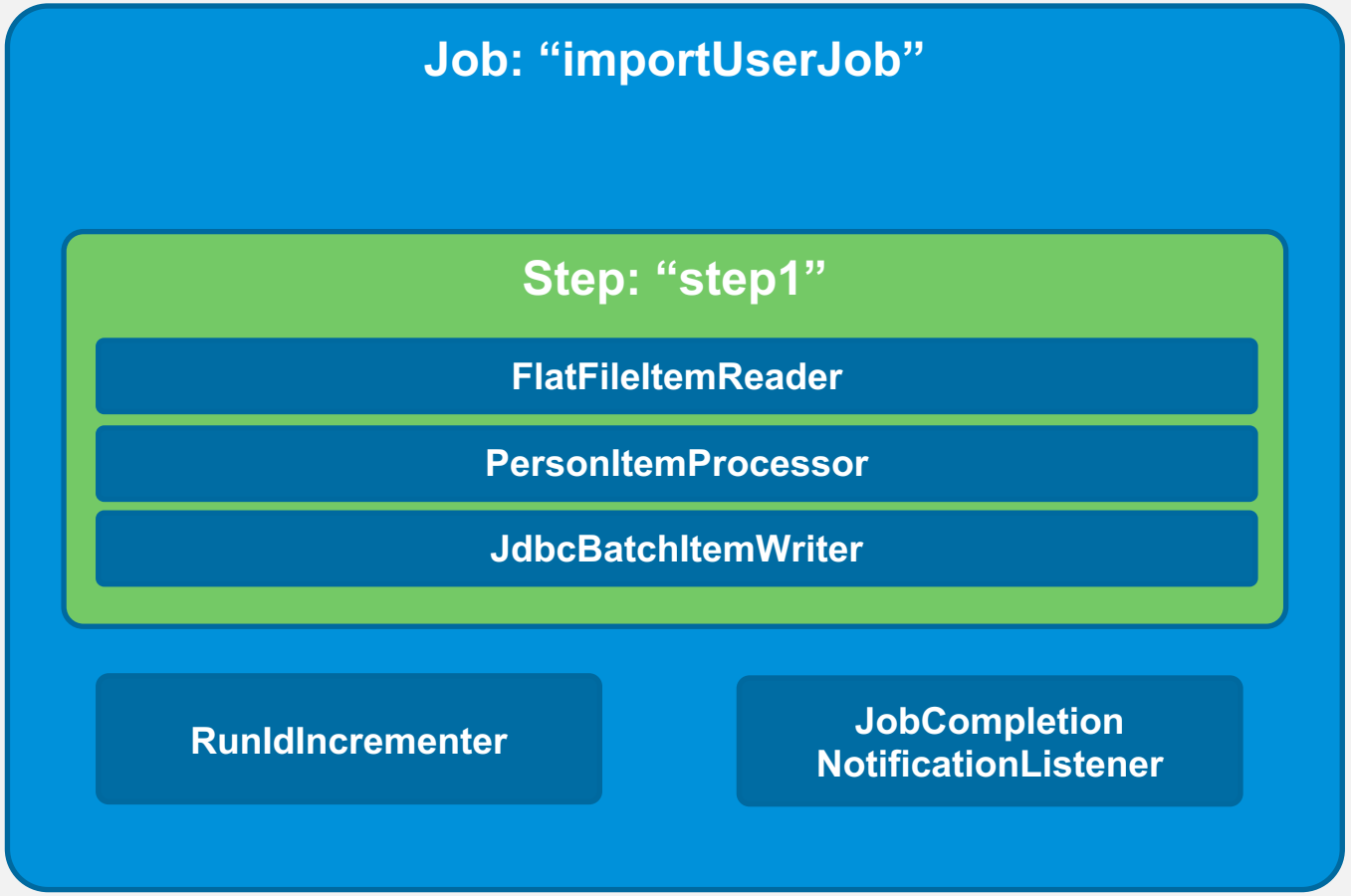
```
@Bean
public Job importUserJob(JobCompletionNotificationListener listener, Step step1)
{
  return jobBuilderFactory.get("importUserJob")
    .incrementer(new RunIdIncrementer())
    .listener(listener)
    .flow(step1)
    .end()
    .build();
}

@Bean
public Step step1(JdbcBatchItemWriter<Person> writer) {
  return stepBuilderFactory.get("step1")
    .<Person, Person> chunk(10)
    .reader(reader())
    .processor(processor())
    .writer(writer)
    .build();
}
```

COPY



Recap

Job, Step, Reader, Processor, Writer



Jobs have several additional properties which can be used to control their behavior.

- Restartability

- Before and After Job/Step execution

- Id generation strategy

- Exit code mapping

- Line aggregation

- Writing headers and footers

- …

Wiring Things Up

Execution Listener

Prints out the results of running the Job once the Job has completed

```
@Component
public class JobCompletionNotificationListener extends
JobExecutionListenerSupport {

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

  private final JdbcTemplate jdbcTemplate;

  @Autowired
  public JobCompletionNotificationListener(JdbcTemplate jdbcTemplate) {
    this.jdbcTemplate = jdbcTemplate;
  }

  @Override
  public void afterJob(JobExecution jobExecution) {
    if(jobExecution.getStatus() == BatchStatus.COMPLETED) {
      log.info("!!! JOB FINISHED! Time to verify the results");

      jdbcTemplate.query("SELECT first_name, last_name FROM people",
        (rs, row) -> new Person(
          rs.getString(1),
          rs.getString(2))
      ).forEach(person -> log.info("Found <" + person + "> in the database."));
    }
  }
}
```

Wiring Things Up

The Application

Create a runnable JAR file

COPY

```
package com.example.batchprocessing;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class BatchProcessingApplication {

  public static void main(String[] args) throws Exception {
    SpringApplication.run(BatchProcessingApplication.class, args);
  }
}
```

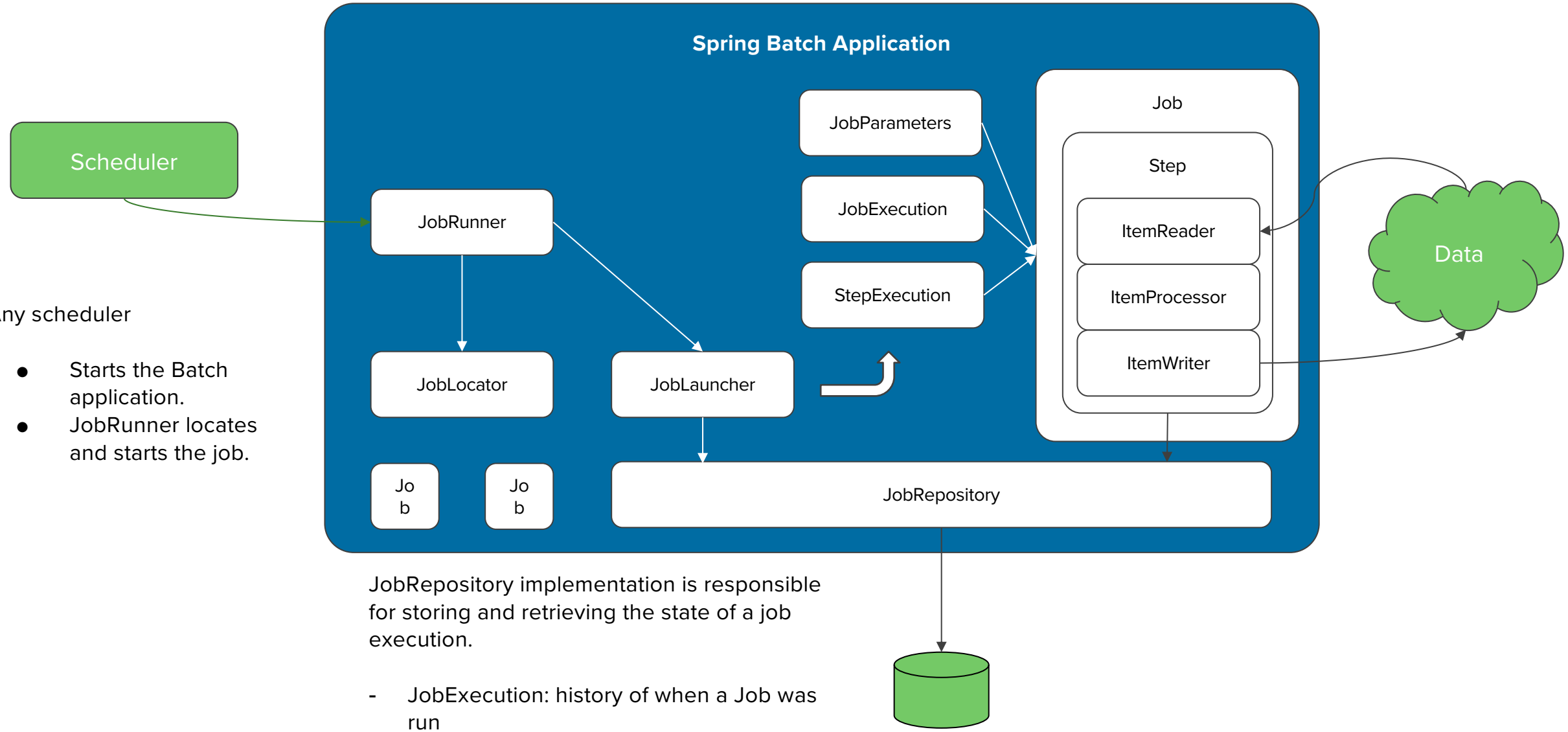
Results of Running The Job

Not Just Another Hello World!

```
Converting (firstName: Jill, lastName: Doe) into (firstName: JILL, lastName:
Converting (firstName: Joe, lastName: Doe) into (firstName: JOE, lastName: DOE)
Converting (firstName: Justin, lastName: Doe) into (firstName: JUSTIN, lastName:
DOE)
Converting (firstName: Jane, lastName: Doe) into (firstName: JANE, lastName: DOE)
Converting (firstName: John, lastName: Doe) into (firstName: JOHN, lastName: DOE)
Found <firstName: JILL, lastName: DOE> in the database.
Found <firstName: JOE, lastName: DOE> in the database.
Found <firstName: JUSTIN, lastName: DOE> in the database.
Found <firstName: JANE, lastName: DOE> in the database.
Found <firstName: JOHN, lastName: DOE> in the database.
```

COPY

More Terminology



Any scheduler

- Starts the Batch application.
- JobRunner locates and starts the job.

JobRepository implementation is responsible for storing and retrieving the state of a job execution.

- JobExecution: history of when a Job was run
- StepExecution: history of Steps in a Job

JobExecution Properties

Property	Definition
Status	A BatchStatus object that indicates the status of the execution. While running, it is BatchStatus#STARTED. If it fails, it is BatchStatus#FAILED. If it finishes successfully, it is BatchStatus#COMPLETED
startTime	A java.util.Date representing the current system time when the execution was started. This field is empty if the job has yet to start.
endTime	A java.util.Date representing the current system time when the execution finished, regardless of whether or not it was successful. The field is empty if the job has yet to finish.
exitStatus	The ExitStatus, indicating the result of the run. It is most important, because it contains an exit code that is returned to the caller. See chapter 5 for more details. The field is empty if the job has yet to finish.
createTime	A java.util.Date representing the current system time when the JobExecution was first persisted. The job may not have been started yet (and thus has no start time), but it always has a createTime, which is required by the framework for managing job level ExecutionContexts.
lastUpdated	A java.util.Date representing the last time a JobExecution was persisted. This field is empty if the job has yet to start.
executionContext	The "property bag" containing any user data that needs to be persisted between executions.
failureExceptions	The list of exceptions encountered during the execution of a Job. These can be useful if more than one exception is encountered during the failure of a Job.

StepExecution Properties

Property	Definition
Status	A BatchStatus object that indicates the status of the execution. While running, the status is BatchStatus.STARTED. If it fails, the status is BatchStatus.FAILED. If it finishes successfully, the status is BatchStatus.COMPLETED.
startTime	A java.util.Date representing the current system time when the execution was started. This field is empty if the step has yet to start.
endTime	A java.util.Date representing the current system time when the execution finished, regardless of whether or not it was successful. This field is empty if the step has yet to exit.
exitStatus	The ExitStatus indicating the result of the execution. It is most important, because it contains an exit code that is returned to the caller. See chapter 5 for more details. This field is empty if the job has yet to exit.
executionContext	The "property bag" containing any user data that needs to be persisted between executions.
readCount	The number of items that have been successfully read.
writeCount	The number of items that have been successfully written.
commitCount	The number of transactions that have been committed for this execution.
rollbackCount	The number of times the business transaction controlled by the Step has been rolled back.
readSkipCount	The number of times read has failed, resulting in a skipped item.
processSkipCount	The number of times process has failed, resulting in a skipped item.
filterCount	The number of items that have been 'filtered' by the ItemProcessor.
writeSkipCount	The number of times write has failed, resulting in a skipped item.

Questions?

Thank You