SPRING BATCH

banuprakashc@yahoo.co.in

Scheduling in Spring with Quartz

- Key Components of the Quartz API
- Quartz has a modular architecture. It consists of several basic components that can be combined as required
- Although we will use Spring to manage the application, each individual component can configured in two ways: the Quartz way or the Spring way.

MethodInvokingJobDetailFactoryBean

- Spring provides MethodInvokingJobDetailFactoryBean that exposes org.quartz.JobDetail.
- While creating job class, it will not extend any quartz job interface and we need to configure the executing method name to JavaConfig.
- Create bean for MethodInvokingJobDetailFactoryBean and configure job class name and method name to execute.
- To provide job class, we have three methods.
  - setTargetBeanName(): Accepts job bean name. Use it if our job class is spring managed bean.
  - setTargetClass() : Accepts class name. Use it if the job class has static method to execute.
  - setTargetObject() : Accepts job object. Use it if job class has no static method to execute.

Job and MethodInvokingJobDetailFactoryBean

```
/**
 *
 * @author Banu Prakash
 *
 */
@Service("jobone")
public class MyJobOne {
    protected void myTask() {
        System.out.println("This is my task");
    }
}
```

```
@Bean
public MethodInvokingJobDetailFactoryBean methodInvokingJobDetailFactoryBean() {
    MethodInvokingJobDetailFactoryBean obj =
            new MethodInvokingJobDetailFactoryBean();
    obj.setTargetBeanName("jobone");
    obj.setTargetMethod("myTask");
    return obj;
}
```

SimpleTriggerFactoryBean

- The spring API SimpleTriggerFactoryBean uses org.quartz.SimpleTrigger.
- Create a bean for it in JavaConfig and configure JobDetail using MethodInvokingJobDetailFactoryBean.
- Configure start delay, repeat interval, repeat count etc as required.

```
// Job is scheduled for 3+1 times with the interval of 30 seconds
@Bean
public SimpleTriggerFactoryBean simpleTriggerFactoryBean() {
    SimpleTriggerFactoryBean stFactory = new SimpleTriggerFactoryBean();
    stFactory.setJobDetail(methodInvokingJobDetailFactoryBean().getObject());
    stFactory.setStartDelay(3000);
    stFactory.setRepeatInterval(30000);
    stFactory.setRepeatCount(3);
    return stFactory;
}
```

JobDetailFactoryBean

- Spring provides JobDetailFactoryBean that uses org.quartz.JobDetail.
- We use it to configure complex job such as job scheduling using cron-expression.
- Create job implementing QuartzJobBean interface and configure to JobDetailFactoryBean.
- We also configure job name and group name.
- To pass the parameter to job, it provides setJobDataAsMap() method.

```
@Bean
public JobDetailFactoryBean jobDetailFactoryBean() {
    JobDetailFactoryBean factory = new JobDetailFactoryBean();
    factory.setJobClass(MyJobTwo.class);
    Map<String, Object> map = new HashMap<String, Object>();
    map.put("name", "BANU");
    map.put(MyJobTwo.COUNT, 1);
    factory.setJobDataAsMap(map);
    factory.setGroup("mygroup");
    factory.setName("myjob");
    return factory;
}
```

QuartzJobBean

- Job implementing QuartzJobBean interface
- executeInternal() is called when the job is scheduled using CronTriggerFactoryBean.
- If we want to persist the changes in JobDataMap, use @PersistJobDataAfterExecution
- If there is more than one trigger which are scheduling same job then to avoid race condition use @DisallowConcurrentExecution

```
@PersistJobDataAfterExecution
@DisallowConcurrentExecution
public class MyJobTwo extends QuartzJobBean {
    public static final String COUNT = "count";
    private String name;

    @Override
    public void executeInternal(JobExecutionContext ctx)
            throws JobExecutionException {
        JobDataMap dataMap = ctx.getJobDetail().getJobDataMap();
        int cnt = dataMap.getInt(COUNT);
        JobKey jobKey = ctx.getJobDetail().getKey();
        System.out.println(jobKey + ": " + name + ": " + cnt);
        cnt++;
        dataMap.put(COUNT, cnt);
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

CronTriggerFactoryBean

- Spring provides CronTriggerFactoryBean that uses org.quartz.CronTrigger.
- CronTriggerFactoryBean configures JobDetailFactoryBean .
- We also configure start delay, trigger name, group name and cron-expression to schedule the job.

```
// Job is scheduled after every 1 minute
@Bean
public CronTriggerFactoryBean cronTriggerFactoryBean() {
    CronTriggerFactoryBean stFactory = new CronTriggerFactoryBean();
    stFactory.setJobDetail(jobDetailFactoryBean().getObject());
    stFactory.setStartDelay(3000);
    stFactory.setName("mytrigger");
    stFactory.setGroup("mygroup");
    stFactory.setCronExpression("0 0/1 * 1/1 * ? *");
    return stFactory;
}
```

Cron-Expressions

1. Seconds
2. Minutes
3. Hours
4. Day-of-Month
5. Month
6. Day-of-Week
7. Year (optional field)

SchedulerFactoryBean

- Spring provides SchedulerFactoryBean that uses org.quartz.Scheduler.
- Using SchedulerFactoryBean we register all the triggers.
- In our case we have two triggers SimpleTriggerFactoryBean and CronTriggerFactoryBean that are being registered.

```
@Bean
public SchedulerFactoryBean schedulerFactoryBean() {
    SchedulerFactoryBean scheduler = new SchedulerFactoryBean();
    scheduler.setTriggers(simpleTriggerFactoryBean().getObject(),
                          cronTriggerFactoryBean().getObject());
    return scheduler;
}
```

Main Class to Test Application

```java
@Configuration
@ComponentScan(basePackages = "com.banu")
public class Main {
    public static void main(String[] args) {
        // SpringApplication.run(Main.class, args);
        AnnotationConfigApplicationContext ctx =
                new AnnotationConfigApplicationContext();
        ctx.register(Main.class);
        ctx.refresh();
    }
}
```

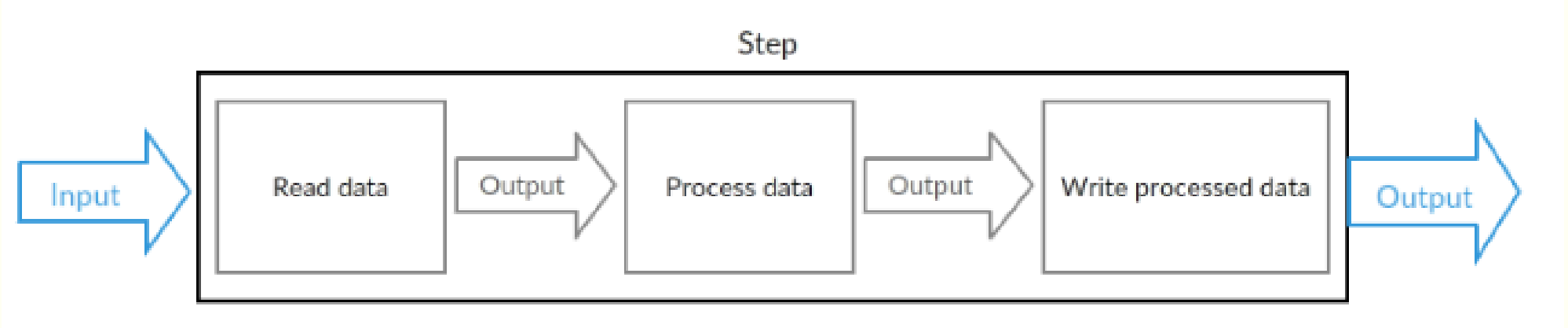
```
This is my task
This is my task
mygroup.myjob: BANU: 1
This is my task
This is my task
mygroup.myjob: BANU: 2
mygroup.myjob: BANU: 3
mygroup.myjob: BANU: 4
```

Batch Processing

- “Batch processing is defined as the processing of data without interaction or interruption“.
- Typically Batch Jobs are long-running, non-interactive and process large volumes of data, more than fits in memory or a single transaction. Thus they usually run outside office hours and include logic for handling errors and restarting if necessary

What are batch applications?

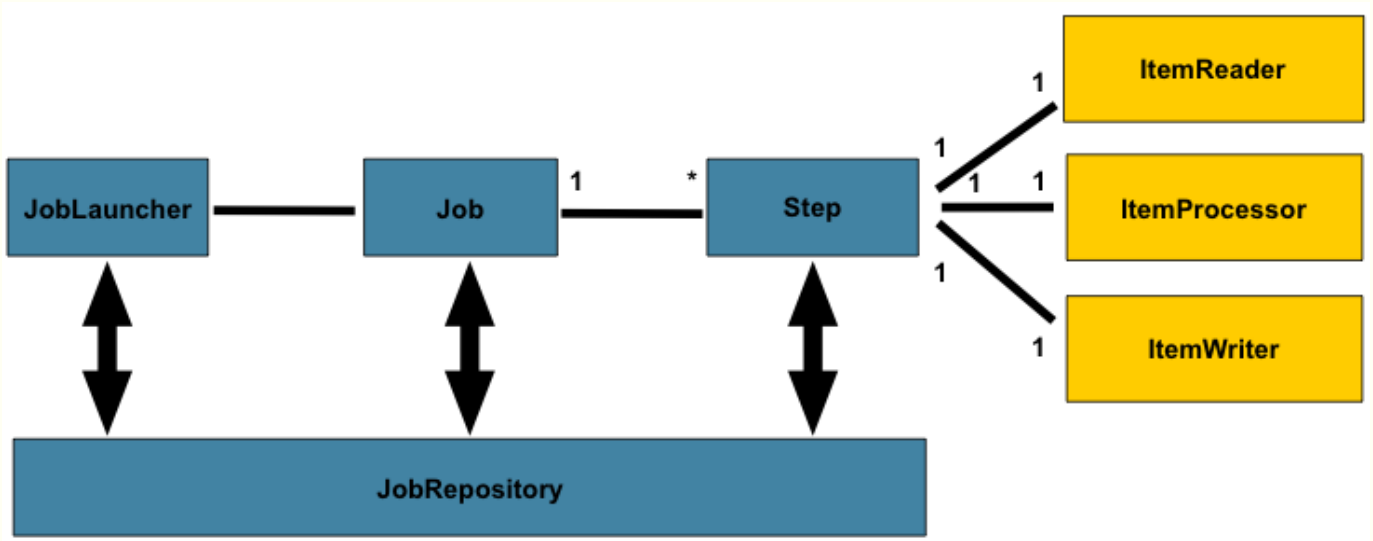
- Batch applications process large amounts of data without human intervention.
- You'd opt to use batch applications to compute data for generating monthly financial statements, calculating statistics, and indexing files.
- A typical batch application: read input data, processes the input data, and write the processed data to the configured output



Spring Batch features

- Transaction management, to allow you to focus on business processing.
- Chunk based processing, to process a large value of data by dividing it in small pieces.
- Start/Stop/Restart/Skip/Retry capabilities, to handle non-interactive management of the process.
- Web based administration interface (Spring Batch Admin), it provides an API for administering tasks.
- Based on Spring framework, so it includes all the configuration options, including Dependency Injection.
- Compliance with JSR 352: Batch Applications for the Java Platform.

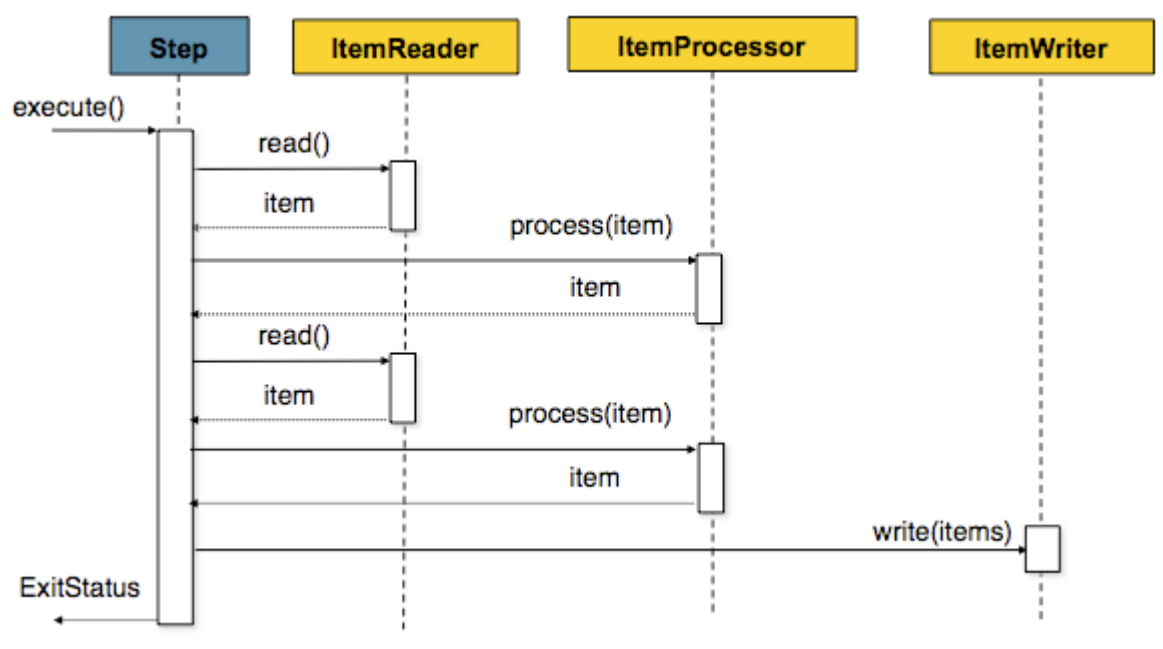
Spring Batch concepts



- Job: an entity that encapsulates an entire batch process. It is composed of one or more ordered **Steps** and it has some properties such as restartability.
- Step: a domain object that encapsulates an independent, sequential phase of a batch job.
- Item: the individual piece of data that it's been processed.

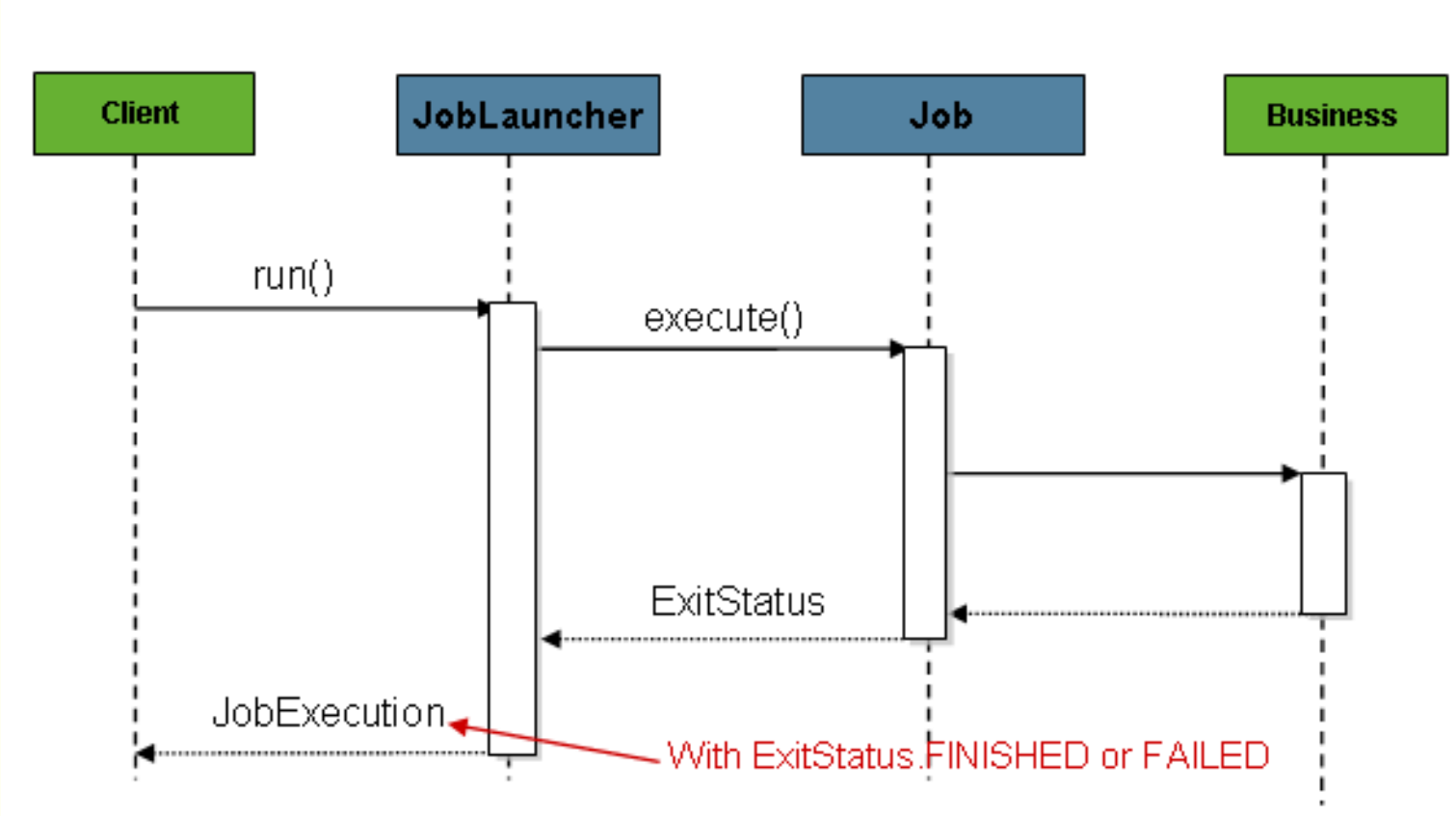
Spring Batch concepts

- Chunk: the processing style used by Spring Batch: read and process the item and then aggregate until reach a number of items, called "*chunk*" that will be finally written.
- JobLauncher: the entry point to launch Spring Batch jobs with a given set of JobParameters.
- JobRepository: maintains all metadata related to job executions and provides CRUD operations for JobLauncher, Job, and Step implementations.



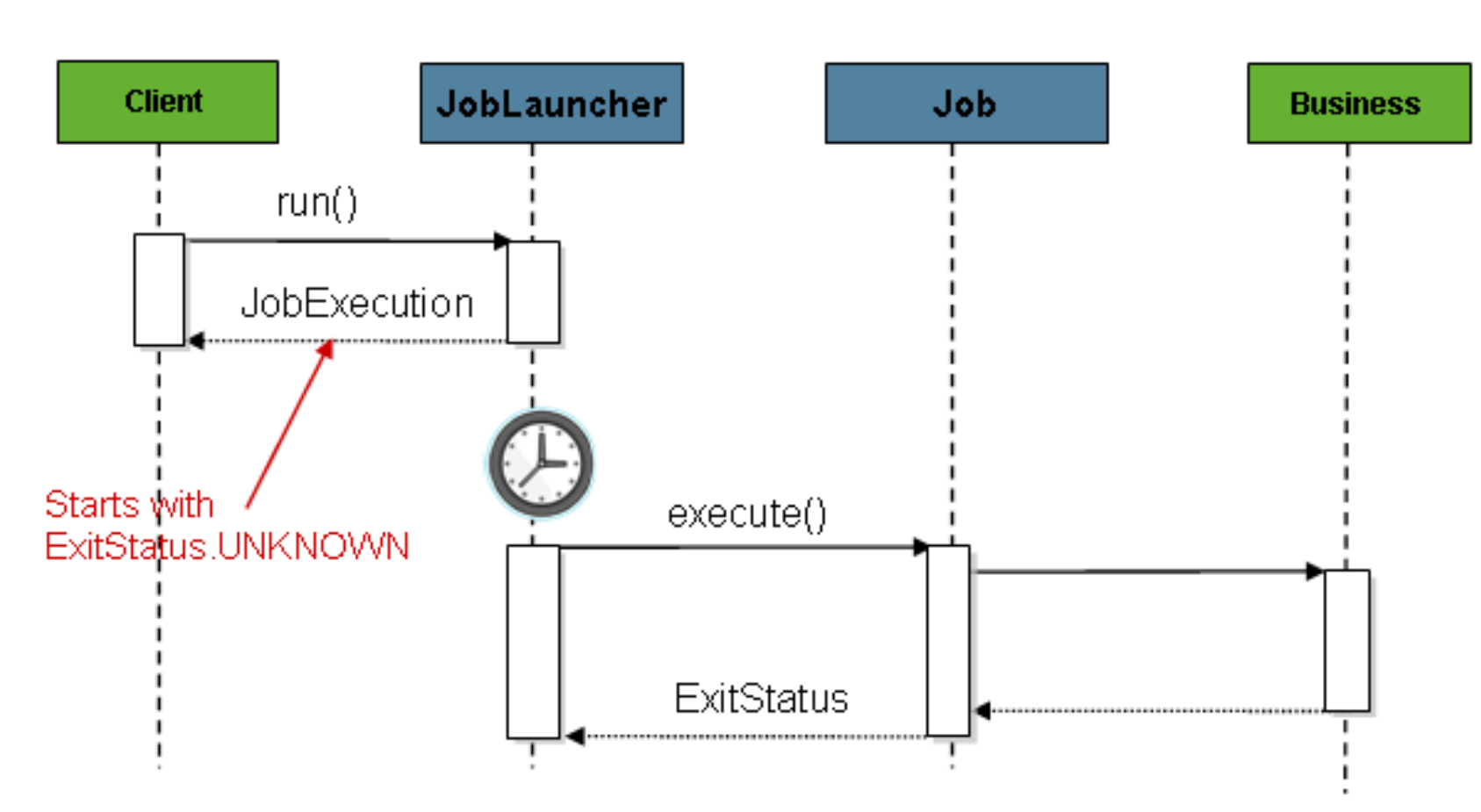
Running a Job

- The JobLauncher interface has a basic implementation SimpleJobLauncher whose only required dependency is a JobRepository, in order to obtain an execution, so that you can use it for executing the Job



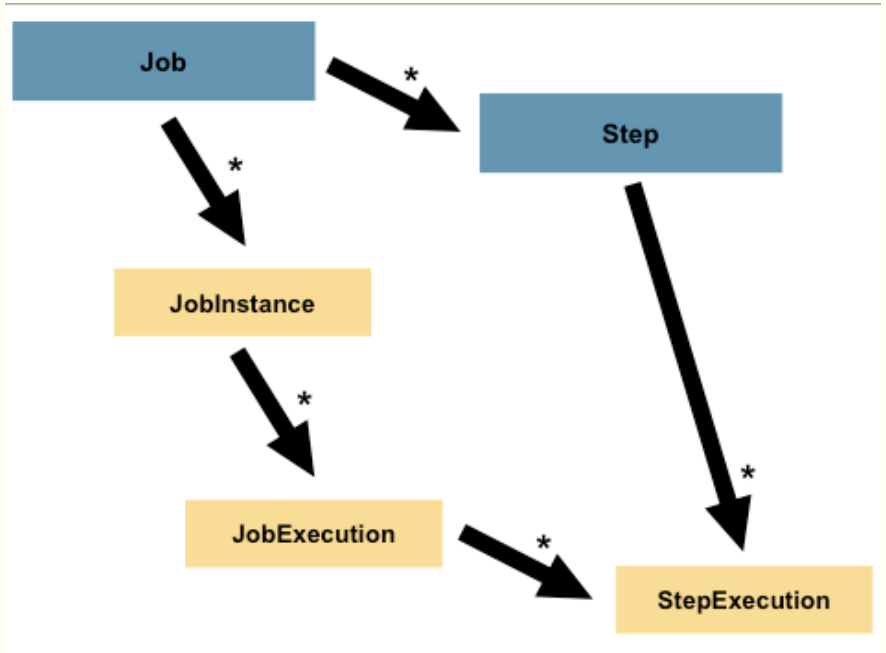
Running a Job asynchronously

- You can also launch a Job asynchronously by configuring a TaskExecutor



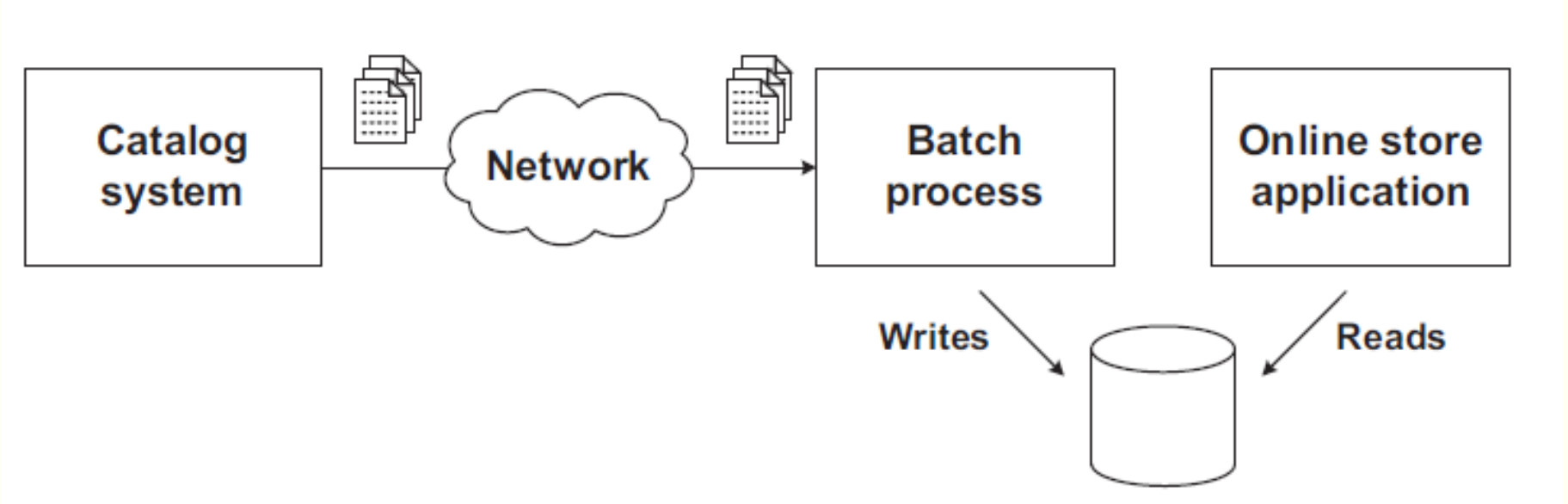
Running Jobs: concepts

- JobInstance: a logical run of a Job.
- JobParameters: a set of parameters used to start a batch job. It categorizes each JobInstance.
- JobExecution: physical runs of Jobs, in order to know what happens with the execution.
- StepExecution: a single attempt to execute a Step, that is created each time a Step is run and it also provides information regarding the result of the processing.
- ExecutionContext: a collection of key/value pairs that are persisted and controlled by the framework in order to allow developers a place to store persistent state that is scoped to a StepExecution or JobExecution.



Spring Batch Example

- ACME Corporation will use batch jobs to populate the online store database with the catalog from its internal proprietary system. The system will process data every night to insert new products in the catalog or update existing products.



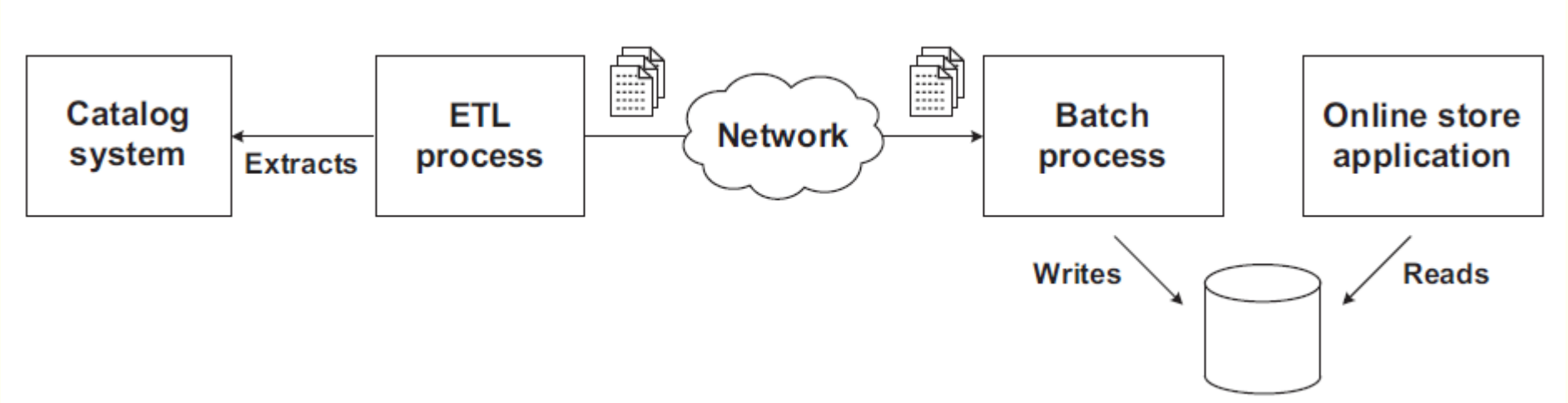
Why build an online store with batch jobs?

- Why did ACME choose to shuttle data from one system to the other instead of making its onsite catalog and the online store communicate directly?
- The software that powers the catalog has an API, so why not use it?
  - The main reason is security: ACME's own network hosts the catalog system, and the company doesn't want to expose the catalog system to the outside world directly, even via another application.
  - Another reason for this architecture is that the catalog system's API and data format don't suit the needs of the online store application: ACME wants to show a summarized view of the catalog data to customers without overwhelming them with a complex catalog structure and supplying too many details. You could get this summarized catalog view by using the catalog system's API, but you'd need to make many calls, which would cause performance to suffer in the catalog system.
- To summarize, a mismatch exists between the view of the data provided by the catalog system and the view of the data required by the online store application. Therefore, an application needs to process the data before exposing it to customers through the online store.

banuprakashc@yahoo.co.in

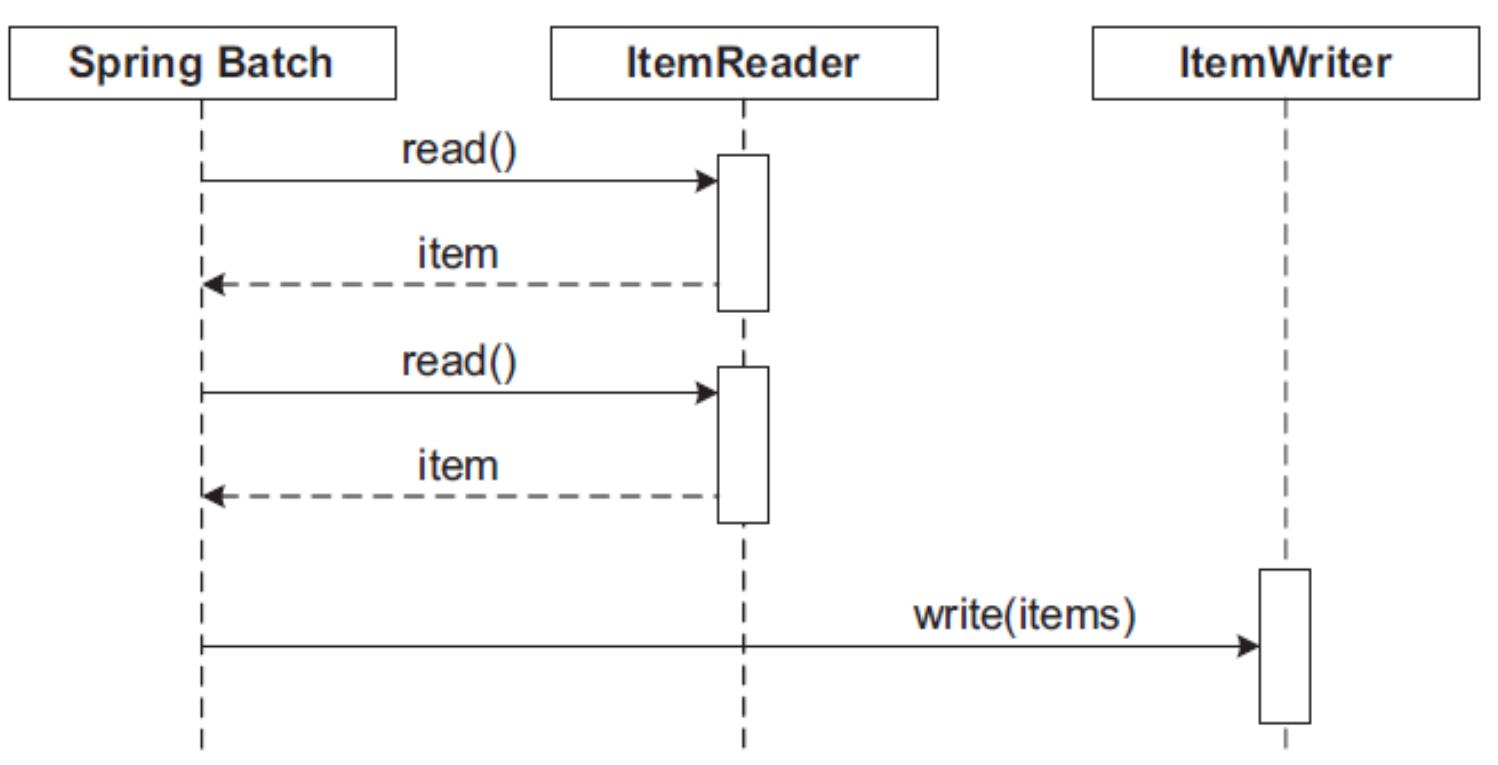
Why use batch processes?

- ACME updates the catalog system throughout the day, adding new products and updating existing products.
- The online store application doesn’t need to expose live data because buyers can live with day-old catalog information
- Therefore, a nightly batch process updates the online store database, using flat files



Read-Write Scenario

- Spring Batch reads items one by one from an **ItemReader**, collects the items in a chunk of a given size, and sends that chunk to an **ItemWriter**

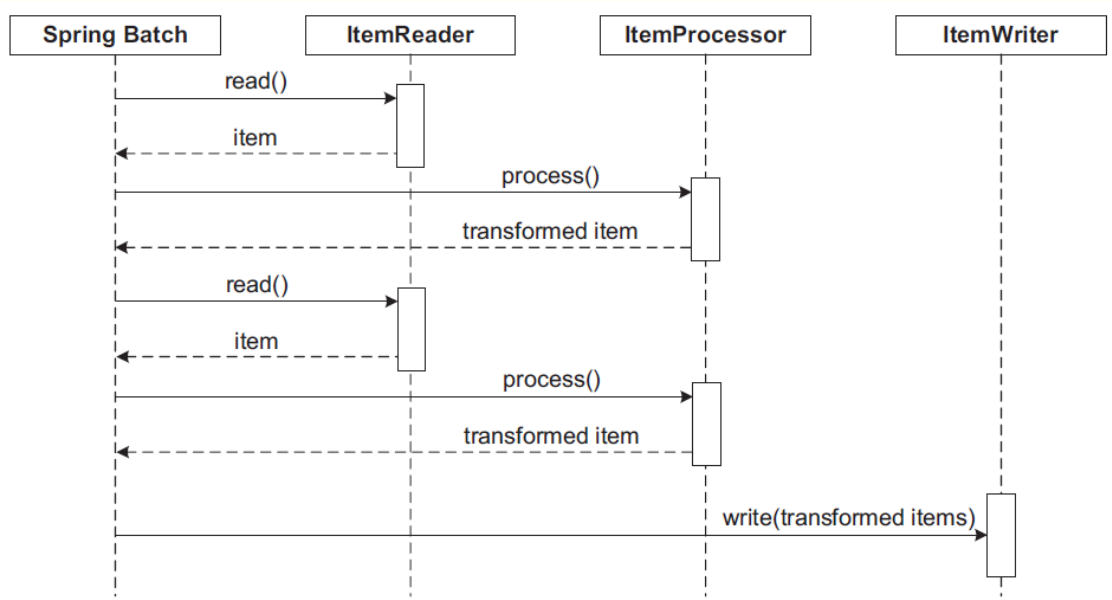


CHUNK PROCESSING

- Chunk processing is particularly well suited to handle large data operations because a job handles items in small chunks instead of processing them all at once.
- Practically speaking, a large file won’t be loaded in memory; instead it’s streamed, which is more efficient in terms of memory consumption.
- Chunk processing allows more flexibility to manage the data flow in a job.
- Spring Batch also handles transactions and errors around read and write operations.

ItemProcessor

- You can perform any transformations you need on an item before Spring Batch sends it to the ItemWriter. This is where you implement the logic to transform the data from the input format into the format expected by the target system.
- Spring Batch also lets you validate and filter input items. If you return null from the ItemProcessor method process, processing for that item stops and Spring Batch won't insert the item in the database



Spring Batch interfaces for chunk processing

```
package org.springframework.batch.item;

public interface ItemReader<T> {

  T read() throws Exception, UnexpectedInputException,
                  ParseException,
                  NonTransientResourceException;

}
```

Reads item

```
package org.springframework.batch.item;

public interface ItemProcessor<I, O> {

  O process(I item) throws Exception;

}
```

Transforms item (optional)

```
package org.springframework.batch.item;

import java.util.List;

public interface ItemWriter<T> {

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

}
```

Writes a chunk of items

Example using Annotation

- @EnableBatchProcessing provides a base configuration for building batch jobs. Within this base configuration, an instance of StepScope is created in addition to a number of beans made available to be autowired:
  - JobRepository - bean name "jobRepository"
  - JobLauncher - bean name "jobLauncher"
  - JobRegistry - bean name "jobRegistry"
  - PlatformTransactionManager - bean name "transactionManager"
  - JobBuilderFactory - bean name "jobBuilders"
  - StepBuilderFactory - bean name "stepBuilders"
- The core interface for this configuration is the BatchConfigurer. The default implementation provides the beans mentioned above and requires a DataSource as a bean within the context to be provided. This data source will be used by the JobRepository.

Configure Reader, Processor and Writer

- ItemReader

```
@Bean
public ItemReader<Product> reader() {
    FlatFileItemReader<Product> reader = new FlatFileItemReader<Product>();
    reader.setResource(new ClassPathResource("products.txt"));
    reader.setLineMapper(new DefaultLineMapper<Product>() {{
        setLineTokenizer(new DelimitedLineTokenizer() {{
            setNames(new String[] {"id", "name","description","price"});
        }});
        setFieldSetMapper(new BeanWrapperFieldSetMapper<Product>() {{
            setTargetType(Product.class);
        }});

    }});
    return reader;
}
```

Configure Reader, Processor and Writer

- Processor and Writer

```
@Bean
public ItemProcessor<Product, Product> processor() {
    return new ProductItemProcessor();
}

@Bean
public ItemWriter<Product> writer(DataSource dataSource) {
    final String INSERT_PRODUCT = "insert into product "
            + "(id,name,description,price) values(:id,:name,:description,:price)";

    JdbcBatchItemWriter<Product> writer = new JdbcBatchItemWriter<Product>();
    writer.setItemSqlParameterSourceProvider(new BeanPropertyItemSqlParameterSourceProvider<Product>());
    writer.setSql(INSERT_PRODUCT);
    writer.setDataSource(dataSource);
    return writer;
}
```

Job configuration

- A Job bean, that it’s built using the JobBuilderFactory that is autowired by passing it as method parameter for this @Bean method.
- When you call its get method, Spring Batch will create a job builder and will initialize its job repository, the JobBuilder is the convenience class for building jobs

```
//  Actual job configuration
@Bean
public Job importProductJob(JobBuilderFactory jobs, Step s1) {
    return jobs.get("importProductJob")
              .incrementer(new RunIdIncrementer())
              .flow(s1)
              .end()
              .build();
}
```

Step configuration

- A Step bean, built using the StepBuilderFactory that is autowired by passing it as method parameter for this @Bean method, as well as the other dependencies: the reader, the processor and the writer previously defined.
- When calling the get method from the StepBuilderFactory, Spring Batch will create a step builder and will initialize its job repository and transaction manager, the StepBuilder is an entry point for building all kinds of steps as you can see in the code.

```
@Bean
public Step step1(StepBuilderFactory stepBuilderFactory, ItemReader<Product> reader,
        ItemWriter<Product> writer, ItemProcessor<Product, Product> processor) {
    return stepBuilderFactory.get("step1")
            .<Product, Product> chunk(10)
            .reader(reader)
            .processor(processor)
            .writer(writer)
            .build();
}
```

DataSource and JdbcTemplate

```
@Bean
public JdbcTemplate jdbcTemplate(DataSource dataSource) {
    return new JdbcTemplate(dataSource);
}
```

```
@Configuration
public class DataSourceConfiguration {

    @Bean
    public DataSource dataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
         dataSource.setDriverClassName("com.mysql.jdbc.Driver");
         dataSource.setUrl("jdbc:mysql://localhost:3306/batch_db");
         dataSource.setUsername("root");
         dataSource.setPassword("Welcome123");
        return dataSource;
    }
}
```

JobLauncher

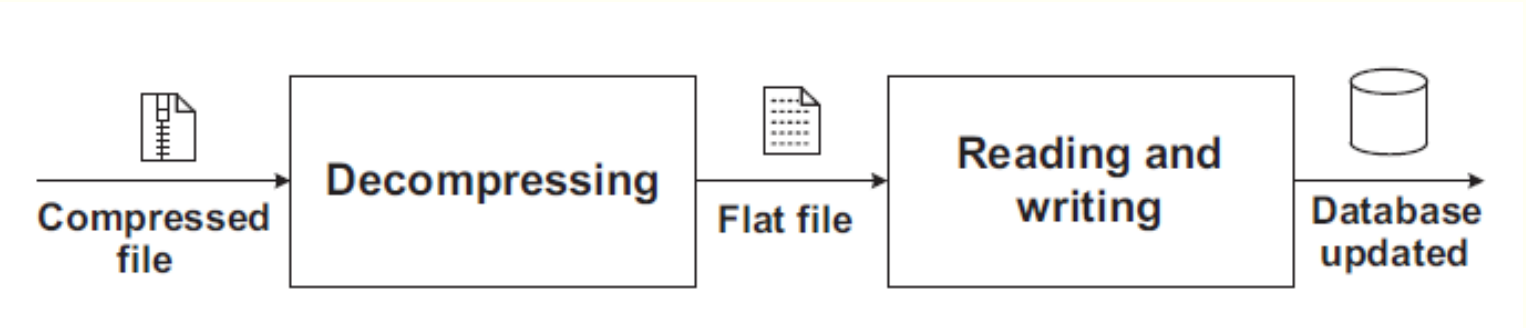
```
@Component
public class MainJobLauncher {

    @Autowired
    JobLauncher jobLauncher;
    @Autowired
    Job importProductJob;

    public static void main(String... args)
            throws JobParametersInvalidException, JobExecutionAlreadyRunningException,
            JobRestartException, JobInstanceAlreadyCompleteException {

        AnnotationConfigApplicationContext context =
                new AnnotationConfigApplicationContext(ApplicationConfiguration.class);

        MainJobLauncher main = context.getBean(MainJobLauncher.class);
        JobExecution jobExecution = main.jobLauncher.run(main.importProductJob, new JobParameters());
        MainHelper.reportResults(jobExecution);
        MainHelper.reportProduct(context.getBean(JdbcTemplate.class));
        context.close();
    }
}
```

The import product use case

- The import product batch reads the product records from a flat file created by ACME and updates the online store application database accordingly.

- **Decompression**
  - Decompresses the archive flat file received from the ACME network. The file is compressed to speed up transfer over the internet.

- Reading and writing
  - The flat file is read line by line and then inserted into the database.



Spring Batch features introduced by the import product job

- Decompression
  - Custom processing in a job (but not reading from a data store and writing to another)
- Read-write
  - Reading a flat file
  - Implementing a custom database writing component
  - Skipping invalid records instead of failing the whole process
- Configuration
  - Leveraging Spring’s lightweight container and Spring Batch’s namespace to wire up batch components
  - Using the Spring Expression Language to make the configuration more flexible

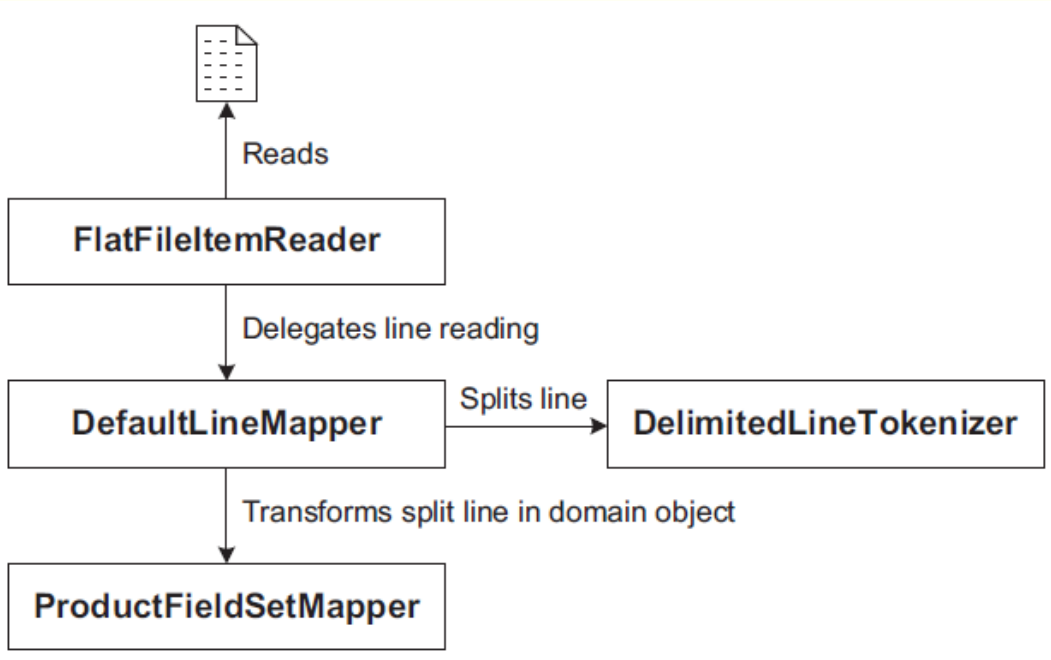
banuprakashc@yahoo.co.in

Reading and writing the product data

- THE FLAT FILE FORMAT
  - PRODUCT_ID,NAME,DESCRIPTION,PRICE
  - PR....210,BlackBerry 8100 Pearl,A cell phone,124.60
  - PR....211,Sony Ericsson W810i,Yet another cell phone!,139.45
  - PR....212,Samsung MM-A900M Ace,A cell phone,97.80
  - PR....213,Toshiba M285-E 14,A cell phone,166.20
  - PR....214,Nokia 2610 Phone,A cell phone,145.50

Reading and writing the product data

- The **FlatFileItemReader** reads the flat file and delegates the mapping between a line and a domain object to a **LineMapper**.
- The **LineMapper** implementation delegates the splitting of lines and the mapping between split lines and domain objects.



CONFIGURATION OF THE FLATFILEITEMREADER

- The FlatFileItemReader can be configured like any Spring bean using an XML configuration file

```
<bean id="reader" class="org.springframework.batch.item.file.FlatFileItemReader">
    <property name="resource"
            value="file:./input/products.txt" />
    <property name="linesToSkip" value="1" />
    <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="PRODUCT_ID,NAME,DESCRIPTION,PRICE" />
                </bean>
            </property>
            <property name="fieldSetMapper">
                <bean class="com.banu.batch.ProductFieldSetMapper" />
            </property>
        </bean>
    </property>
</bean>
```

FIELDSETMAPPER FOR PRODUCT OBJECTS

- FieldSetMapper to convert the line split by the LineTokenizer into a domain object.
- ProductFieldSetMapper focuses on retrieving the data from the flat file and converts values into Product domain objects

```
/**
 * @author Banu Prakash
 *
 */
public class ProductFieldSetMapper implements FieldSetMapper<Product> {

     * (non-Javadoc)
    public Product mapFieldSet(FieldSet fieldSet) throws BindException {
        Product product = new Product();
        product.setId(fieldSet.readString("PRODUCT_ID"));
        product.setName(fieldSet.readString("NAME"));
        product.setDescription(fieldSet.readString("DESCRIPTION"));
        product.setPrice(fieldSet.readBigDecimal("PRICE"));
        return product;
    }

}
```

Implementing a database item writer

```
public class ProductJdbcItemWriter implements ItemWriter<Product> {

    private JdbcTemplate jdbcTemplate;

    private static final String INSERT_PRODUCT = "insert into product "
            + "(id,name,description,price) values(?,?,?,?)";
    private static final String UPDATE_PRODUCT = "update product set name=?, "
            + " description=?, price=? where id = ?";

    public ProductJdbcItemWriter(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

    public void write(List<? extends Product> items) throws Exception {
        for(Product item : items) {
            int updated = jdbcTemplate.update(UPDATE_PRODUCT,
                item.getName(),item.getDescription(),item.getPrice(),item.getId());
            if(updated == 0) {
                jdbcTemplate.update(
                    INSERT_PRODUCT,
                    item.getId(),item.getName(),item.getDescription(),item.getPrice());
            }
        }
    }
}
```

```
<bean id="writer" class="com.banu.batch.ProductJdbcItemWriter">
    <constructor-arg ref="dataSource" />
</bean>
```

Configuring the read-write step

- The commit-interval attribute is set to a chunk size. Recommendation is a value between 10 and 200.
- Too small a chunk size creates too many transactions, which is costly and makes the job run slowly. Too large a chunk size makes transactional resources like databases run slowly too, because a database must be able to roll back operations

```
<job id="importProducts" xmlns="http://www.springframework.org/schema/batch">
    <step id="readWriteProducts">
        <tasklet>
            <chunk reader="reader" writer="writer" commit-interval="3" skip-limit="5">
                <skippable-exception-classes>
                    <include class="org.springframework.batch.item.file.FlatFileParseException" />
                </skippable-exception-classes>
            </chunk>
        </tasklet>
    </step>
</job>
```

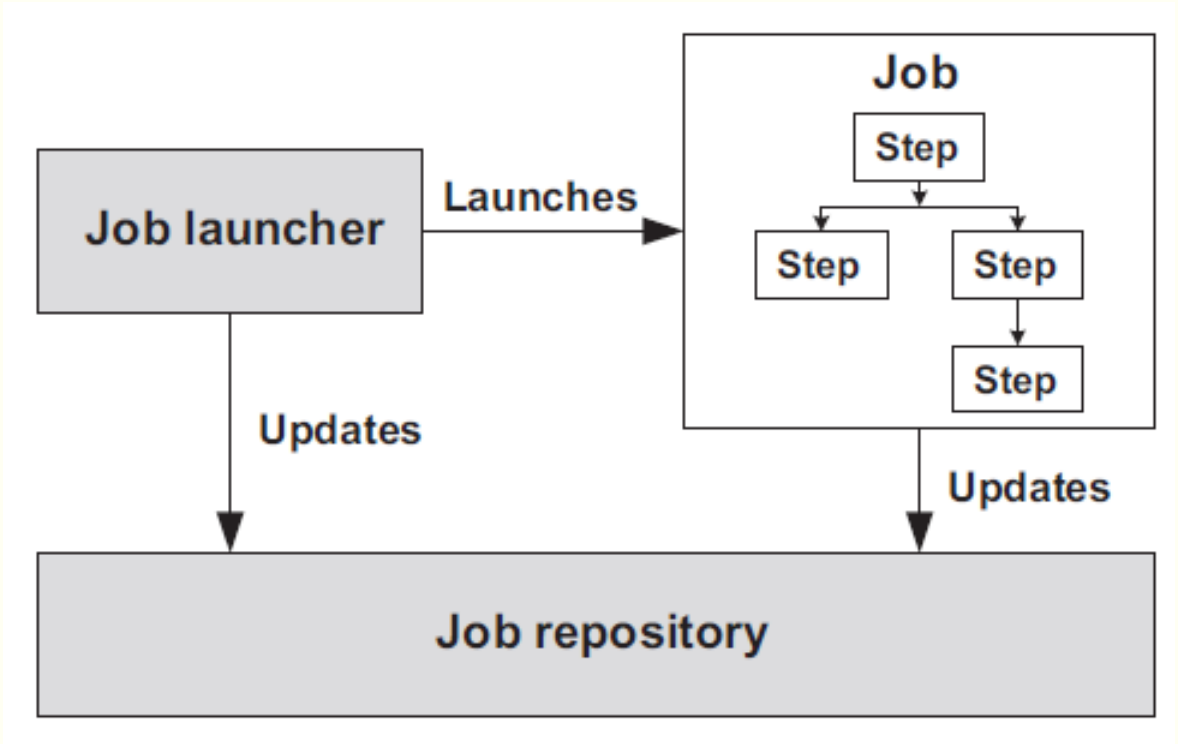
Spring Batch needs infrastructure components

- Setting up the batch infrastructure is a mandatory step for a batch application, which you need to do only once for all jobs living in the same Spring application context.
- The jobs will use the same infrastructure components to run and to store their state. These infrastructure components are the key to managing and monitoring jobs
- Spring Batch needs two infrastructure components:
  - *Job repository*—To store the state of jobs (finished or currently running)
  - *Job launcher*—To create the state of a job before launching it

```
<bean id="jobRepository"
    class="org.springframework.batch.core.repository.support.MapJobRepositoryFactoryBean">
    <property name="transactionManager" ref="transactionManager" />
</bean>

<bean id="jobLauncher"
    class="org.springframework.batch.core.launch.support.SimpleJobLauncher">
    <property name="jobRepository" ref="jobRepository" />
</bean>
```

Spring Batch components

- The infrastructure components provided by Spring Batch are in gray
- Application components implemented by the developer—are in white.

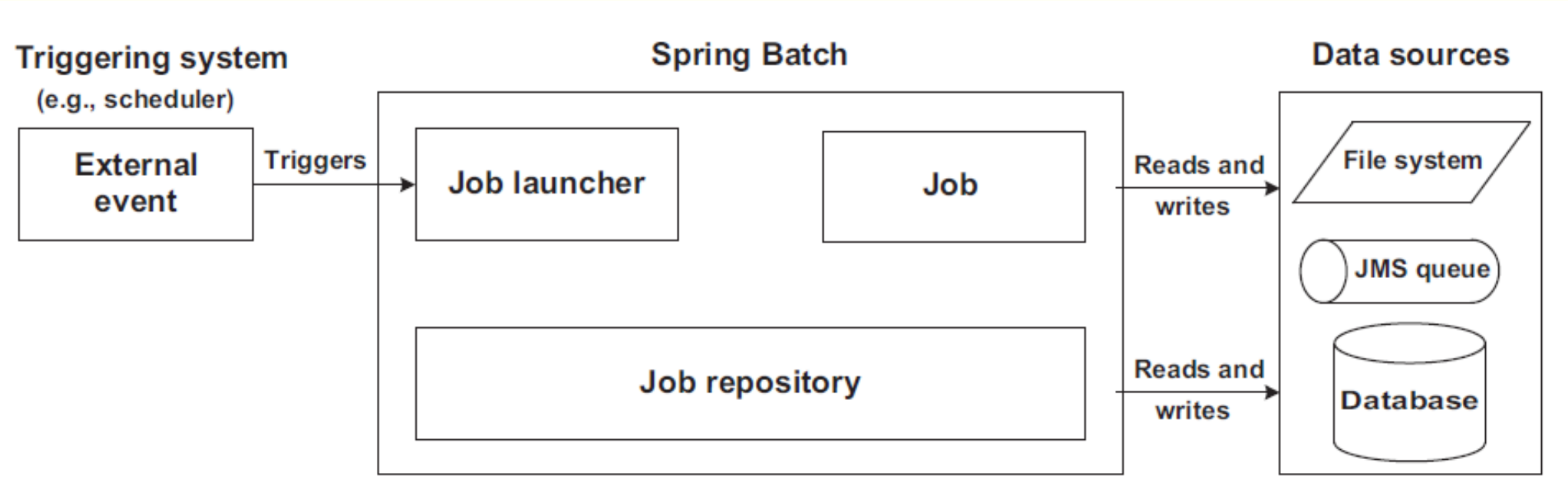


The main components of a Spring Batch application

Component	Description
Job repository	An infrastructure component that persists job execution metadata
Job launcher	An infrastructure component that starts job executions
Job	An application component that represents a batch process
Step	A phase in a job; a job is a sequence of steps
Tasklet	A transactional, potentially repeatable process occurring in a step
Item	A record read from or written to a data source
Chunk	A list of items of a given size
Item reader	A component responsible for reading items from a data source
Item processor	A component responsible for processing (transforming, validating, or filtering) a read item before it's written
Item writer	A component responsible for writing a chunk to a data source

Spring Batch interaction with the outside world

- A job starts in response to an event. The event can come from anywhere: a system scheduler like cron that runs periodically, a script that launches a Spring Batch process, an HTTP request to a web controller that launches the job, and so on.
- Jobs can communicate with data sources, but so does the job repository.

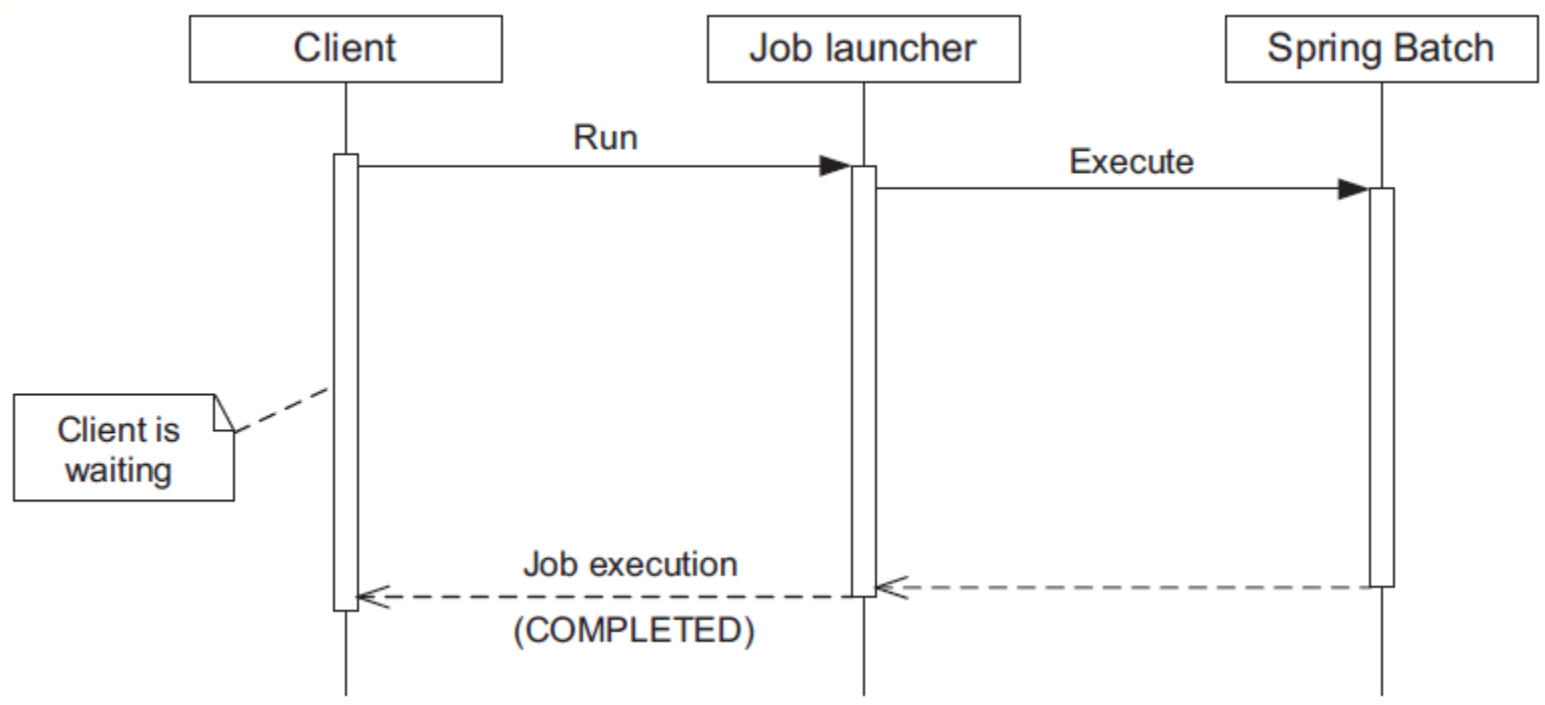


Launching jobs

```
ApplicationContext context = new ClassPathXmlApplicationContext("job-context.xml");

JobLauncher jobLauncher = (JobLauncher) context.getBean("jobLauncher");
Job job = (Job) context.getBean("importProducts");

try {
    JobExecution execution = jobLauncher.run(job, new JobParameters());
    System.out.println("Job Exit Status : " + execution.getStatus());

} catch (JobExecutionException e) {
    System.out.println("Job  failed");
    e.printStackTrace();
}
```

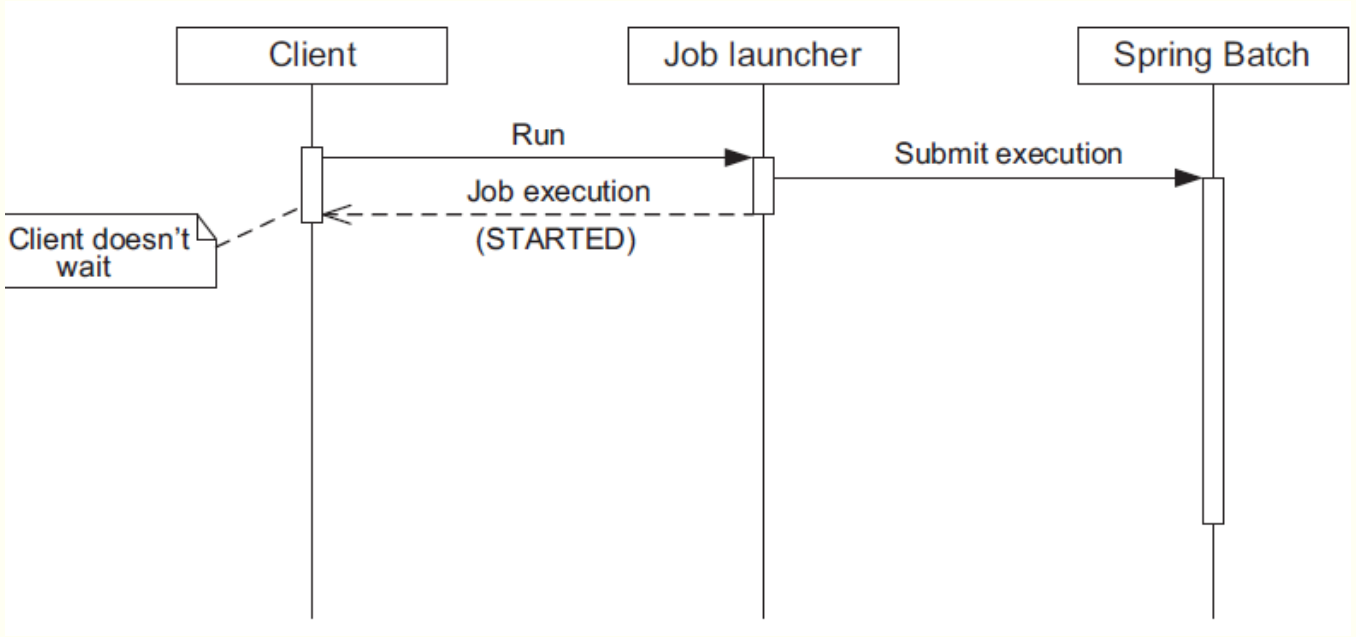
Synchronous launch

- By default, the JobLauncher run method is synchronous: the caller waits until the job execution ends (successfully or not).



Asynchronous

- The job launcher can use a task executor to launch job executions asynchronously.



Asynchronous

- To make the job launcher asynchronous, just provide it with an appropriate TaskExecutor, as shown in the following snippet.
- The executor reuses threads from its pool to launch job executions asynchronously. Note the use of the executor XML element from the task namespace. This is a shortcut provided in Spring 3.0, but you can also define a task executor like any other bean (by using an implementation like ThreadPoolTaskExecutor)

```
<task:executor id="executor" pool-size="10" />

<bean id="jobLauncher" class="org.springframework.
  ➥ batch.core.launch.support.SimpleJobLauncher">
  <property name="jobRepository" ref="jobRepository" />
  <property name="taskExecutor" ref="executor" />
</bean>
```