Table of Contents

Chapter 1. Configuring and Running a Job

In the domain section , the overall architecture design was discussed, using the following diagram as a guide:

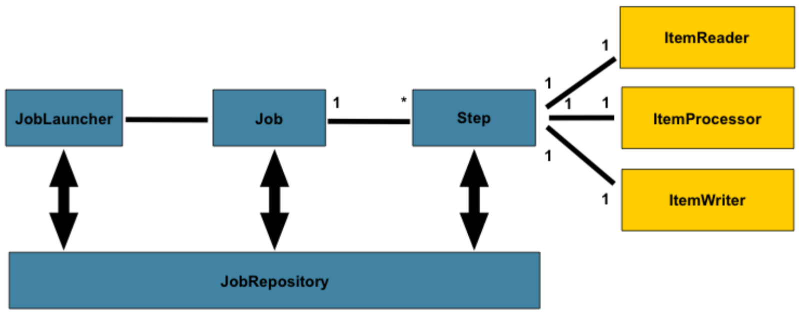

Figure 1. Batch Stereotypes

While the `Job` object may seem like a simple container for steps, there are many configuration options of which a developer must be aware. Furthermore, there are many considerations for how a `Job` will be run and how its meta-data will be stored during that run. This chapter will explain the various configuration options and runtime concerns of a `Job`.

1.1. Configuring a Job

There are multiple implementations of the `Job` interface, however this is abstracted behind either the builders provided for java configuration or the XML namespace when using XML based configuration.

Java Configuration

```
@Bean
public Job footballJob() {
    return this.jobBuilderFactory.get("footballJob")
                     .start(playerLoad())
                     .next(gameLoad())
                     .next(playerSummarization())
                     .end()
                     .build();
}
```

XML Configuration

```
<job id="footballJob">
    <step id="playerload"          parent="s1" next="gameLoad"/>
    <step id="gameLoad"            parent="s2" next="playerSummarization"/>
    <step id="playerSummarization" parent="s3"/>
</job>
```

The examples here use a parent bean definition to create the steps; see the section on step configuration for more options declaring specific step details inline. The XML namespace defaults to referencing a repository with an id of 'jobRepository', which is a sensible default. However, this can be overridden explicitly:

```
<job id="footballJob" job-repository="specialRepository">
    <step id="playerload"          parent="s1" next="gameLoad"/>
    <step id="gameLoad"            parent="s3" next="playerSummarization"/>
    <step id="playerSummarization" parent="s3"/>
</job>
```

In addition to steps a job configuration can contain other elements that help with parallelisation (`<split>`), declarative flow control (`<decision>`) and externalization of flow definitions (`<flow/>`).

1.1.1. Restartability

One key issue when executing a batch job concerns the behavior of a `Job` when it is restarted. The launching of a `Job` is considered to be a 'restart' if a `JobExecution` already exists for the particular `JobInstance`. Ideally, all jobs should be able to start up where they left off, but there are scenarios where this is not possible. *It is entirely up to the developer to ensure that a new* `JobInstance` *is created in this scenario*. However, Spring Batch does provide some help. If a `Job` should never be restarted, but should always be run as part of a new `JobInstance`, then the restartable property may be set to 'false':

XML Configuration

```
<job id="footballJob" restartable="false">
    ...
</job>
```

Java Configuration

```
@Bean
public Job footballJob() {
    return this.jobBuilderFactory.get("footballJob")
                     .preventRestart()
                     ...
                     .build();
}
```

To phrase it another way, setting restartable to false means "this `Job` does not support being started again". Restarting a `Job` that is not restartable will cause a `JobRestartException` to be thrown:

```
Job job = new SimpleJob();
job.setRestartable(false);

JobParameters jobParameters = new JobParameters();

JobExecution firstExecution = jobRepository.createJobExecution(job, jobParameters);
jobRepository.saveOrUpdate(firstExecution);

try {
    jobRepository.createJobExecution(job, jobParameters);
    fail();
}
catch (JobRestartException e) {
    // expected
}
```

This snippet of JUnit code shows how attempting to create a `JobExecution` the first time for a non restartable job will cause no issues. However, the second attempt will throw a `JobRestartException`.

1.1.2. Intercepting Job Execution

During the course of the execution of a Job, it may be useful to be notified of various events in its lifecycle so that custom code may be executed. The `SimpleJob` allows for this by calling a `JobListener` at the appropriate time:

```
public interface JobExecutionListener {

    void beforeJob(JobExecution jobExecution);

    void afterJob(JobExecution jobExecution);

}
```

`JobListeners` can be added to a `SimpleJob` via the listeners element on the job:

XML Configuration

```
<job id="footballJob">
    <step id="playerload"          parent="s1" next="gameLoad"/>
    <step id="gameLoad"            parent="s2" next="playerSummarization"/>
    <step id="playerSummarization" parent="s3"/>
    <listeners>
        <listener ref="sampleListener"/>
    </listeners>
</job>
```

Java Configuration

```
@Bean
public Job footballJob() {
    return this.jobBuilderFactory.get("footballJob")
                     .listener(sampleListener())
                     ...
                     .build();
}
```

It should be noted that `afterJob` will be called regardless of the success or failure of the Job. If success or failure needs to be determined it can be obtained from the `JobExecution`:

```
public void afterJob(JobExecution jobExecution){
    if( jobExecution.getStatus() == BatchStatus.COMPLETED ){
        //job success
    }
    else if(jobExecution.getStatus() == BatchStatus.FAILED){
        //job failure
    }
}
```

The annotations corresponding to this interface are:

- `@BeforeJob`
- `@AfterJob`

1.1.3. Inheriting from a Parent Job

If a group of Jobs share similar, but not identical, configurations, then it may be helpful to define a "parent" `Job` from which the concrete Jobs may inherit properties. Similar to class inheritance in Java, the "child" `Job` will combine its elements and attributes with the parent's.

In the following example, "baseJob" is an abstract `Job` definition that defines only a list of listeners. The `Job` "job1" is a concrete definition that inherits the list of listeners from "baseJob" and merges it with its own list of listeners to produce a `Job` with two listeners and one `Step`, "step1".

```
<job id="baseJob" abstract="true">
    <listeners>
        <listener ref="listenerOne"/>
    <listeners>
</job>

<job id="job1" parent="baseJob">
    <step id="step1" parent="standaloneStep"/>

    <listeners merge="true">
        <listener ref="listenerTwo"/>
    <listeners>
</job>
```

Please see the section on Inheriting from a Parent Step for more detailed information.

This section only applies to XML based configuration as java configuration provides better reuse capabilities.

1.1.4. JobParametersValidator

A job declared in the XML namespace or using any subclass of `AbstractJob` can optionally declare a validator for the job parameters at runtime. This is useful when for instance you need to assert that a job is started with all its mandatory parameters. There is a `DefaultJobParametersValidator` that can be used to constrain combinations of simple mandatory and optional parameters, and for more complex constraints you can implement the interface yourself.

The configuration of a validator is supported through the java builders, e.g:

```
@Bean
public Job job1() {
    return this.jobBuilderFactory.get("job1")
                     .validator(parametersValidator())
                     ...
                     .build();
}
```

XML namespace support is also available for configuration of a `JobParametersValidator`:

```
<job id="job1" parent="baseJob3">
    <step id="step1" parent="standaloneStep"/>
    <validator ref="parametersValidator"/>
</job>
```

The validator can be specified as a reference (as above) or as a nested bean definition in the beans namespace.

1.2. Java Config

Spring 3 brought the ability to configure applications via java in addition to XML. As of Spring Batch 2.2.0, batch jobs can be configured using the same java config. There are two components for the java based configuration: the `@EnableBatchProcessing` annotation and two builders.

The `@EnableBatchProcessing` works similarly to the other @Enable* annotations in the Spring family. In this case, `@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. You can customize any of these beans by creating a custom implementation of the `BatchConfigurer` interface. Typically, extending the `DefaultBatchConfigurer` (which is provided if a `BatchConfigurer` is not found) and overriding the required getter is sufficient. However, implementing your own from scratch may be required. The following example shows how to provide a custom transaction manager:

```
@Bean
public BatchConfigurer batchConfigurer() {
    return new DefaultBatchConfigurer() {
        @Override
        public PlatformTransactionManager getTransactionManager() {
            return new MyTransactionManager();
        }
    };
}
```

Only one configuration class needs to have the `@EnableBatchProcessing` annotation. Once you have a class annotated with it, you will have all of the above available.

With the base configuration in place, a user can use the provided builder factories to configure a job. Below is an example of a two step job configured via the `JobBuilderFactory` and the `StepBuilderFactory`.

```java
@Configuration
@EnableBatchProcessing
@Import(DataSourceConfiguration.class)
public class AppConfig {

    @Autowired
    private JobBuilderFactory jobs;

    @Autowired
    private StepBuilderFactory steps;

    @Bean
    public Job job(@Qualifier("step1") Step step1, @Qualifier("step2") Step step2) {
        return jobs.get("myJob").start(step1).next(step2).build();
    }

    @Bean
    protected Step step1(ItemReader<Person> reader,
                         ItemProcessor<Person, Person> processor,
                         ItemWriter<Person> writer) {
        return steps.get("step1")
            .<Person, Person> chunk(10)
            .reader(reader)
            .processor(processor)
            .writer(writer)
            .build();
    }

    @Bean
    protected Step step2(Tasklet tasklet) {
        return steps.get("step2")
            .tasklet(tasklet)
            .build();
    }
}
```

1.3. Configuring a JobRepository

When using `@EnableBatchProcessing`, a `JobRepository` is provided out of the box for you. This section addresses configuring your own.

As described in earlier, the `JobRepository` is used for basic CRUD operations of the various persisted domain objects within Spring Batch, such as `JobExecution` and `StepExecution`. It is required by many of the major framework features, such as the `JobLauncher`, `Job`, and `Step`.

The batch namespace abstracts away many of the implementation details of the `JobRepository` implementations and their collaborators. However, there are still a few configuration options available:

XML Configuration

```
<job-repository id="jobRepository"
    data-source="dataSource"
    transaction-manager="transactionManager"
    isolation-level-for-create="SERIALIZABLE"
    table-prefix="BATCH_"
    max-varchar-length="1000"/>
```

None of the configuration options listed above are required except the id. If they are not set, the defaults shown above will be used. They are shown above for awareness purposes. The `max-varchar-length` defaults to 2500, which is the length of the long `VARCHAR` columns in the sample schema scripts

When using java configuration, a `JobRepository` is provided for you. A JDBC based one is provided out of the box if a `DataSource` is provided, the `Map` based one if not. However you can customize the configuration of the `JobRepository` via an implementation of the `BatchConfigurer` interface.

Java Configuration

```
...
// This would reside in your BatchConfigurer implementation
@Override
protected JobRepository createJobRepository() throws Exception {
    JobRepositoryFactoryBean factory = new JobRepositoryFactoryBean();
    factory.setDataSource(dataSource);
    factory.setTransactionManager(transactionManager);
    factory.setIsolationLevelForCreate("ISOLATION_SERIALIZABLE");
    factory.setTablePrefix("BATCH_");
    factory.setMaxVarCharLength(1000);
    return factory.getObject();
}
...
```

None of the configuration options listed above are required except the dataSource and transactionManager. If they are not set, the defaults shown above will be used. They are shown above for awareness purposes. The max varchar length defaults to 2500, which is the length of the long `VARCHAR` columns in the sample schema scripts

1.3.1. Transaction Configuration for the JobRepository

If the namespace or the provided `FactoryBean` is used, transactional advice will be automatically created around the repository. This is to ensure that the batch meta data, including state that is necessary for restarts after a failure, is persisted correctly. The behavior of the framework is not well defined if the repository methods are not transactional. The isolation level in the `create*` method attributes is specified separately to ensure that when jobs are launched, if two processes are trying to launch the same job at the same time, only one will succeed. The default isolation level for that method is SERIALIZABLE, which is quite aggressive: READ_COMMITTED would work just as well; READ_UNCOMMITTED would be fine if two processes are not likely to collide in this way.

However, since a call to the `create*` method is quite short, it is unlikely that the SERIALIZED will cause problems, as long as the database platform supports it. However, this can be overridden:

XML Configuration

```
<job-repository id="jobRepository"
                isolation-level-for-create="REPEATABLE_READ" />
```

Java Configuration

```
// This would reside in your BatchConfigurer implementation
@Override
protected JobRepository createJobRepository() throws Exception {
    JobRepositoryFactoryBean factory = new JobRepositoryFactoryBean();
    factory.setDataSource(dataSource);
    factory.setTransactionManager(transactionManager);
    factory.setIsolationLevelForCreate("ISOLATION_REPEATABLE_READ");
    return factory.getObject();
}
```

If the namespace or factory beans aren't used then it is also essential to configure the transactional behavior of the repository using AOP:

XML Configuration

```
<aop:config>
    <aop:advisor
           pointcut="execution(* org.springframework.batch.core..*Repository+.*(..))
"/>
    <advice-ref="txAdvice" />
</aop:config>

<tx:advice id="txAdvice" transaction-manager="transactionManager">
    <tx:attributes>
        <tx:method name="*" />
    </tx:attributes>
</tx:advice>
```

This fragment can be used as is, with almost no changes. Remember also to include the appropriate namespace declarations and to make sure spring-tx and spring-aop (or the whole of spring) are on the classpath.

Java Configuration

```
@Bean
public TransactionProxyFactoryBean baseProxy() {
    TransactionProxyFactoryBean transactionProxyFactoryBean = new
TransactionProxyFactoryBean();
    Properties transactionAttributes = new Properties();
    transactionAttributes.setProperty("*", "PROPAGATION_REQUIRED");
    transactionProxyFactoryBean.setTransactionAttributes(transactionAttributes);
    transactionProxyFactoryBean.setTarget(jobRepository());
    transactionProxyFactoryBean.setTransactionManager(transactionManager());
    return transactionProxyFactoryBean;
}
```

1.3.2. Changing the Table Prefix

Another modifiable property of the `JobRepository` is the table prefix of the meta-data tables. By default they are all prefaced with BATCH_. BATCH_JOB_EXECUTION and BATCH_STEP_EXECUTION are two examples. However, there are potential reasons to modify this prefix. If the schema names needs to be prepended to the table names, or if more than one set of meta data tables is needed within the same schema, then the table prefix will need to be changed:

XML Configuration

```
<job-repository id="jobRepository"
                table-prefix="SYSTEM.TEST_" />
```

Java Configuration

```
// This would reside in your BatchConfigurer implementation
@Override
protected JobRepository createJobRepository() throws Exception {
    JobRepositoryFactoryBean factory = new JobRepositoryFactoryBean();
    factory.setDataSource(dataSource);
    factory.setTransactionManager(transactionManager);
    factory.setTablePrefix("SYSTEM.TEST_");
    return factory.getObject();
}
```

Given the above changes, every query to the meta data tables will be prefixed with "SYSTEM.TEST_". BATCH_JOB_EXECUTION will be referred to as SYSTEM.TEST_JOB_EXECUTION.

Only the table prefix is configurable. The table and column names are not.

1.3.3. In-Memory Repository

There are scenarios in which you may not want to persist your domain objects to the database. One reason may be speed; storing domain objects at each commit point takes extra time. Another reason may be that you just don't need to persist status for a particular job. For this reason, Spring batch provides an in-memory Map version of the job repository:

XML Configuration

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

Java Configuration

```
// This would reside in your BatchConfigurer implementation
@Override
protected JobRepository createJobRepository() throws Exception {
    JobRepositoryFactoryBean factory = new JobRepositoryFactoryBean();
    factory.setDataSource(dataSource);
    factory.setTransactionManager(transactionManager);
    factory.setIsolationLevelForCreate("ISOLATION_REPEATABLE_READ");
    return factory.getObject();
}
```

Note that the in-memory repository is volatile and so does not allow restart between JVM instances. It also cannot guarantee that two job instances with the same parameters are launched simultaneously, and is not suitable for use in a multi-threaded Job, or a locally partitioned `Step`. So use the database version of the repository wherever you need those features.

However it does require a transaction manager to be defined because there are rollback semantics within the repository, and because the business logic might still be transactional (e.g. RDBMS access). For testing purposes many people find the `ResourcelessTransactionManager` useful.

1.3.4. Non-standard Database Types in a Repository

If you are using a database platform that is not in the list of supported platforms, you may be able to use one of the supported types, if the SQL variant is close enough. To do this you can use the raw `JobRepositoryFactoryBean` instead of the namespace shortcut and use it to set the database type to the closest match:

XML Configuration

```
<bean id="jobRepository" class="org...JobRepositoryFactoryBean">
    <property name="databaseType" value="db2"/>
    <property name="dataSource" ref="dataSource"/>
</bean>
```

Java Configuration

```java
// This would reside in your BatchConfigurer implementation
@Override
protected JobRepository createJobRepository() throws Exception {
    JobRepositoryFactoryBean factory = new JobRepositoryFactoryBean();
    factory.setDataSource(dataSource);
    factory.setDatabaseType("db2");
    factory.setTransactionManager(transactionManager);
    return factory.getObject();
}
```

(The `JobRepositoryFactoryBean` tries to auto-detect the database type from the `DataSource` if it is not specified.) The major differences between platforms are mainly accounted for by the strategy for incrementing primary keys, so often it might be necessary to override the `incrementerFactory` as well (using one of the standard implementations from the Spring Framework).

If even that doesn't work, or you are not using an RDBMS, then the only option may be to implement the various `Dao` interfaces that the `SimpleJobRepository` depends on and wire one up manually in the normal Spring way.

1.4. Configuring a JobLauncher

When using `@EnableBatchProcessing`, a `JobRegistry` is provided out of the box for you. This section addresses configuring your own.

The most basic implementation of the `JobLauncher` interface is the `SimpleJobLauncher`. Its only required dependency is a `JobRepository`, in order to obtain an execution:

XML Configuration

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

Java Configuration

```java
...
// This would reside in your BatchConfigurer implementation
@Override
protected JobLauncher createJobLauncher() throws Exception {
    SimpleJobLauncher jobLauncher = new SimpleJobLauncher();
    jobLauncher.setJobRepository(jobRepository);
    jobLauncher.afterPropertiesSet();
    return jobLauncher;
}
...
```

Once a JobExecution is obtained, it is passed to the execute method of Job, ultimately returning the `JobExecution` to the caller:

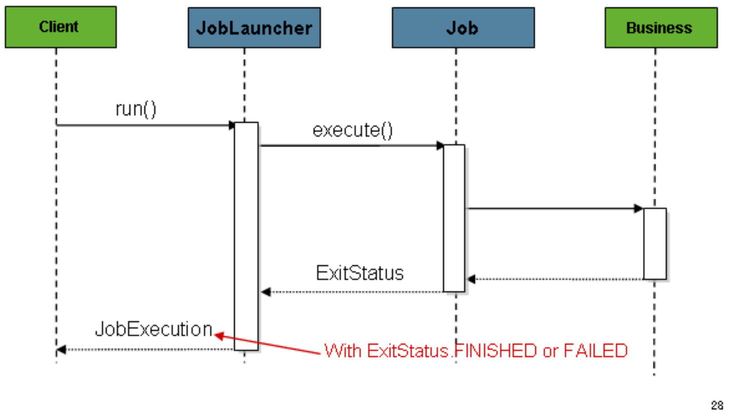

Figure 2. Job Launcher Sequence

The sequence is straightforward and works well when launched from a scheduler. However, issues arise when trying to launch from an HTTP request. In this scenario, the launching needs to be done asynchronously so that the `SimpleJobLauncher` returns immediately to its caller. This is because it is not good practice to keep an HTTP request open for the amount of time needed by long running processes such as batch. An example sequence is below:

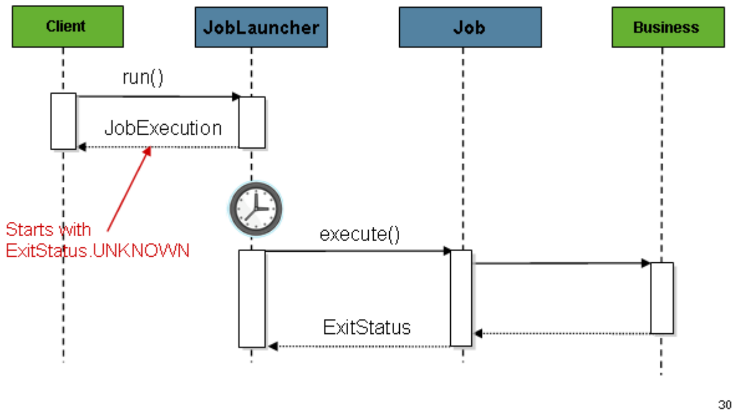

Figure 3. Asynchronous Job Launcher Sequence

The `SimpleJobLauncher` can easily be configured to allow for this scenario by configuring a `TaskExecutor`:

XML Configuration

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

Java Configuration

```
@Bean
public JobLauncher jobLauncher() {
    SimpleJobLauncher jobLauncher = new SimpleJobLauncher();
    jobLauncher.setJobRepository(jobRepository());
    jobLauncher.setTaskExecutor(new SimpleAsyncTaskExecutor());
    jobLauncher.afterPropertiesSet();
    return jobLauncher;
}
```

Any implementation of the spring `TaskExecutor` interface can be used to control how jobs are asynchronously executed.

1.5. Running a Job

At a minimum, launching a batch job requires two things: the `Job` to be launched and a `JobLauncher`. Both can be contained within the same context or different contexts. For example, if launching a job from the command line, a new JVM will be instantiated for each Job, and thus every job will have its own `JobLauncher`. However, if running from within a web container within the scope of an `HttpRequest`, there will usually be one `JobLauncher`, configured for asynchronous job launching, that multiple requests will invoke to launch their jobs.

1.5.1. Running Jobs from the Command Line

For users that want to run their jobs from an enterprise scheduler, the command line is the primary interface. This is because most schedulers (with the exception of Quartz unless using the NativeJob) work directly with operating system processes, primarily kicked off with shell scripts. There are many ways to launch a Java process besides a shell script, such as Perl, Ruby, or even 'build tools' such as ant or maven. However, because most people are familiar with shell scripts, this example will focus on them.

The CommandLineJobRunner

Because the script launching the job must kick off a Java Virtual Machine, there needs to be a class with a main method to act as the primary entry point. Spring Batch provides an implementation that serves just this purpose: `CommandLineJobRunner`. It's important to note that this is just one way to bootstrap your application, but there are many ways to launch a Java process, and this class should in no way be viewed as definitive. The `CommandLineJobRunner` performs four tasks:

- Load the appropriate `ApplicationContext`
- Parse command line arguments into `JobParameters`
- Locate the appropriate job based on arguments
- Use the `JobLauncher` provided in the application context to launch the job.

All of these tasks are accomplished using only the arguments passed in. The following are required arguments:

Table 1. CommandLineJobRunner arguments

jobPath	The location of the XML file that will be used to create an `ApplicationContext`. This file should contain everything needed to run the complete Job
jobName	The name of the job to be run.

These arguments must be passed in with the path first and the name second. All arguments after these are considered to be `JobParameters` and must be in the format of 'name=value':

```
<bash$ java CommandLineJobRunner endOfDayJob.xml endOfDay
schedule.date(date)=2007/05/05
```

```
<bash$ java CommandLineJobRunner io.spring.EndOfDayJobConfiguration endOfDay
schedule.date(date)=2007/05/05
```

In most cases you would want to use a manifest to declare your main class in a jar, but for simplicity, the class was used directly. This example is using the same 'EndOfDay' example from the domainLanguageOfBatch. The first argument is where your job is configured (either an XML file or a fully qualified class name). The second argument, 'endOfDay' represents the job name. The final argument, 'schedule.date(date)=2007/05/05' will be converted into `JobParameters`. An example of the configuration is below:

XML Configuration

```
<job id="endOfDay">
    <step id="step1" parent="simpleStep" />
</job>

<!-- Launcher details removed for clarity -->
<beans:bean id="jobLauncher"
         class="org.springframework.batch.core.launch.support.SimpleJobLauncher" />
```

Java Configuration

```
@Configuration
@EnableBatchProcessing
public class EndOfDayJobConfiguration {

    @Autowired
    private JobBuilderFactory jobBuilderFactory;

    @Autowired
    private StepBuilderFactory stepBuilderFactory;

    @Bean
    public Job endOfDay() {
        return this.jobBuilderFactory.get("endOfDay")
                    .start(step1())
                    .build();
    }

    @Bean
    public Step step1() {
        return this.stepBuilderFactory.get("step1")
                    .tasklet((contribution, chunkContext) -> null)
                    .build();
    }
}
```

This example is overly simplistic, since there are many more requirements to a run a batch job in Spring Batch in general, but it serves to show the two main requirements of the `CommandLineJobRunner`: `Job` and `JobLauncher`

ExitCodes

When launching a batch job from the command-line, an enterprise scheduler is often used. Most schedulers are fairly dumb and work only at the process level. This means that they only know about some operating system process such as a shell script that they're invoking. In this scenario, the only way to communicate back to the scheduler about the success or failure of a job is through return codes. A return code is a number that is returned to a scheduler by the process that indicates the result of the run. In the simplest case: 0 is success and 1 is failure. However, there may be more complex scenarios: If job A returns 4 kick off job B, and if it returns 5 kick off job C. This type of behavior is configured at the scheduler level, but it is important that a processing framework such as Spring Batch provide a way to return a numeric representation of the 'Exit Code' for a particular batch job. In Spring Batch this is encapsulated within an `ExitStatus`, which is covered in more detail in Chapter 5. For the purposes of discussing exit codes, the only important thing to know is that an `ExitStatus` has an exit code property that is set by the framework (or the developer) and is returned as part of the `JobExecution` returned from the `JobLauncher`. The `CommandLineJobRunner` converts this string value to a number using the `ExitCodeMapper` interface:

```
public interface ExitCodeMapper {

    public int intValue(String exitCode);

}
```

The essential contract of an `ExitCodeMapper` is that, given a string exit code, a number representation will be returned. The default implementation used by the job runner is the `SimpleJvmExitCodeMapper` that returns 0 for completion, 1 for generic errors, and 2 for any job runner errors such as not being able to find a `Job` in the provided context. If anything more complex than the 3 values above is needed, then a custom implementation of the `ExitCodeMapper` interface must be supplied. Because the `CommandLineJobRunner` is the class that creates an `ApplicationContext`, and thus cannot be 'wired together', any values that need to be overwritten must be autowired. This means that if an implementation of `ExitCodeMapper` is found within the `BeanFactory`, it will be injected into the runner after the context is created. All that needs to be done to provide your own `ExitCodeMapper` is to declare the implementation as a root level bean and ensure that it is part of the `ApplicationContext` that is loaded by the runner.

1.5.2. Running Jobs from within a Web Container

Historically, offline processing such as batch jobs have been launched from the command-line, as described above. However, there are many cases where launching from an `HttpRequest` is a better option. Many such use cases include reporting, ad-hoc job running, and web application support. Because a batch job by definition is long running, the most important concern is ensuring to launch the job asynchronously:

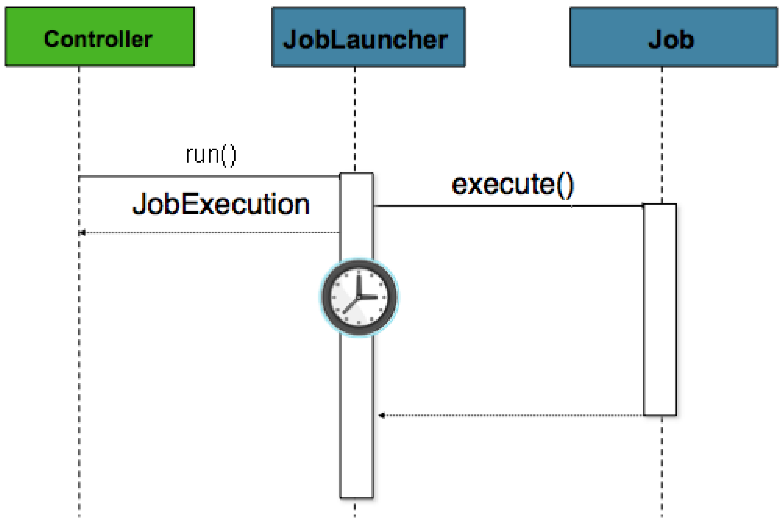

Figure 4. Asynchronous Job Launcher Sequence From Web Container

The controller in this case is a Spring MVC controller. More information on Spring MVC can be found here: https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html#mvc. The controller launches a `Job` using a `JobLauncher` that has been configured to launch asynchronously, which immediately returns a `JobExecution`. The `Job` will likely still be running, however, this nonblocking behaviour allows the controller to return immediately, which is required when handling an `HttpRequest`. An example is below:

```
@Controller
public class JobLauncherController {

    @Autowired
    JobLauncher jobLauncher;

    @Autowired
    Job job;

    @RequestMapping("/jobLauncher.html")
    public void handle() throws Exception{
        jobLauncher.run(job, new JobParameters());
    }
}
```

1.6. Advanced Meta-Data Usage

So far, both the `JobLauncher` and `JobRepository` interfaces have been discussed. Together, they represent simple launching of a job, and basic CRUD operations of batch domain objects:

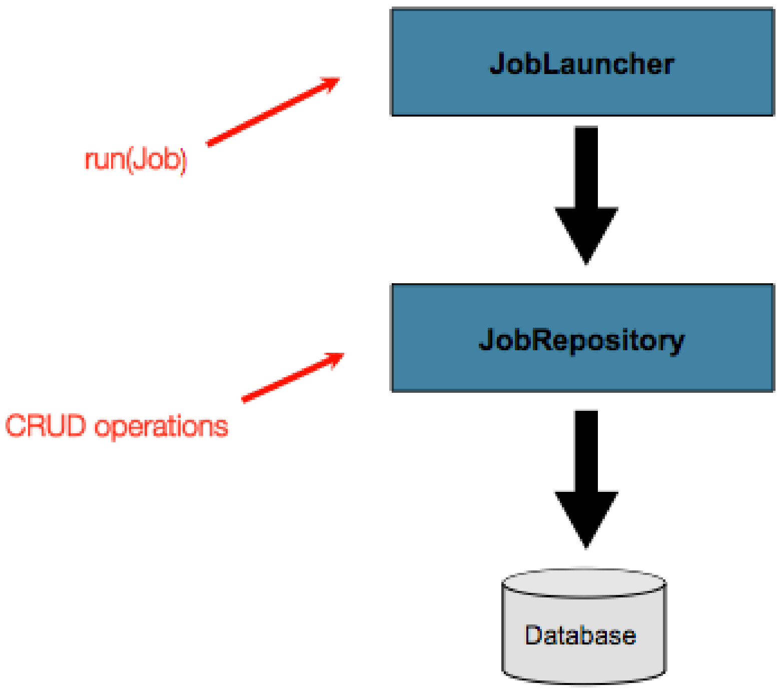

Figure 5. Job Repository

A `JobLauncher` uses the `JobRepository` to create new `JobExecution` objects and run them. `Job` and `Step` implementations later use the same `JobRepository` for basic updates of the same executions during the running of a Job. The basic operations suffice for simple scenarios, but in a large batch environment with hundreds of batch jobs and complex scheduling requirements, more advanced access of the meta data is required:

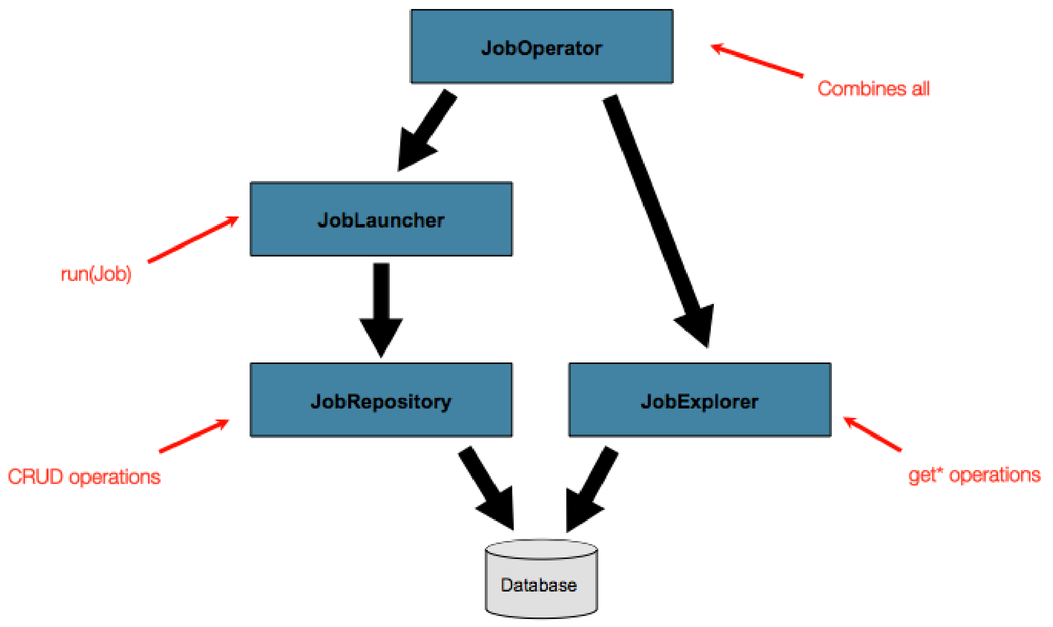

Figure 6. Advanced Job Repository Access

The `JobExplorer` and `JobOperator` interfaces, which will be discussed below, add additional functionality for querying and controlling the meta data.

1.6.1. Querying the Repository

The most basic need before any advanced features is the ability to query the repository for existing executions. This functionality is provided by the `JobExplorer` interface:

```
public interface JobExplorer {

    List<JobInstance> getJobInstances(String jobName, int start, int count);

    JobExecution getJobExecution(Long executionId);

    StepExecution getStepExecution(Long jobExecutionId, Long stepExecutionId);

    JobInstance getJobInstance(Long instanceId);

    List<JobExecution> getJobExecutions(JobInstance jobInstance);

    Set<JobExecution> findRunningJobExecutions(String jobName);
}
```

As is evident from the method signatures above, `JobExplorer` is a read-only version of the `JobRepository`, and like the `JobRepository`, it can be easily configured via a factory bean:

XML Configuration

```
<bean id="jobExplorer" class="org.spr...JobExplorerFactoryBean"
      p:dataSource-ref="dataSource" />
```

Java Configuration

```
...
// This would reside in your BatchConfigurer implementation
@Override
public JobExplorer getJobExplorer() throws Exception {
    JobExplorerFactoryBean factoryBean = new JobExplorerFactoryBean();
    factoryBean.setDataSource(this.dataSource);
    return factoryBean.getObject();
}
...
```

Earlier in this chapter, it was mentioned that the table prefix of the `JobRepository` can be modified to allow for different versions or schemas. Because the `JobExplorer` is working with the same tables, it too needs the ability to set a prefix:

XML Configuration

```
<bean id="jobExplorer" class="org.spr...JobExplorerFactoryBean"
        p:tablePrefix="SYSTEM."/>
```

Java Configuration

```
...
// This would reside in your BatchConfigurer implementation
@Override
public JobExplorer getJobExplorer() throws Exception {
    JobExplorerFactoryBean factoryBean = new JobExplorerFactoryBean();
    factoryBean.setDataSource(this.dataSource);
    factoryBean.setTablePrefix("SYSTEM.");
    return factoryBean.getObject();
}
...
```

1.6.2. JobRegistry

A `JobRegistry` (and its parent interface `JobLocator`) is not mandatory, but it can be useful if you want to keep track of which jobs are available in the context. It is also useful for collecting jobs centrally in an application context when they have been created elsewhere (e.g. in child contexts). Custom `JobRegistry` implementations can also be used to manipulate the names and other properties of the jobs that are registered. There is only one implementation provided by the framework and this is based on a simple map from job name to job instance.

```
<bean id="jobRegistry" class=
"org.springframework.batch.core.configuration.support.MapJobRegistry" />
```

When using `@EnableBatchProcessing`, a `JobRegistry` is provided out of the box for you. If you want to configure your own:

```
...
// This is already provided via the @EnableBatchProcessing but can be customized via
// overriding the getter in the SimpleBatchConfiguration
@Override
@Bean
public JobRegistry jobRegistry() throws Exception {
    return new MapJobRegistry();
}
...
```

There are two ways to populate a `JobRegistry` automatically: using a bean post processor and using a registrar lifecycle component. These two mechanisms are described in the following sections.

JobRegistryBeanPostProcessor

This is a bean post-processor that can register all jobs as they are created:

XML Configuration

```
<bean id="jobRegistryBeanPostProcessor" class="org.spr...JobRegistryBeanPostProcessor
">
    <property name="jobRegistry" ref="jobRegistry"/>
</bean>
```

Java Configuration

```
@Bean
public JobRegistryBeanPostProcessor jobRegistryBeanPostProcessor() {
    JobRegistryBeanPostProcessor postProcessor = new JobRegistryBeanPostProcessor();
    postProcessor.setJobRegistry(jobRegistry());
    return postProcessor;
}
```

Although it is not strictly necessary, the post-processor in the example has been given an id so that it can be included in child contexts (e.g. as a parent bean definition) and cause all jobs created there to also be registered automatically.

AutomaticJobRegistrar

This is a lifecycle component that creates child contexts and registers jobs from those contexts as they are created. One advantage of doing this is that, while the job names in the child contexts still have to be globally unique in the registry, their dependencies can have "natural" names. So for example, you can create a set of XML configuration files each having only one Job, but all having different definitions of an ItemReader with the same bean name, e.g. "reader". If all those files were imported into the same context, the reader definitions would clash and override one another, but with the automatic registrar this is avoided. This makes it easier to integrate jobs contributed from separate modules of an application.

XML Configuration

```xml
<bean class="org.spr...AutomaticJobRegistrar">
   <property name="applicationContextFactories">
      <bean class="org.spr...ClasspathXmlApplicationContextsFactoryBean">
         <property name="resources" value="classpath*:/config/job*.xml" />
      </bean>
   </property>
   <property name="jobLoader">
      <bean class="org.spr...DefaultJobLoader">
         <property name="jobRegistry" ref="jobRegistry" />
      </bean>
   </property>
</bean>
```

Java Configuration

```java
@Bean
public AutomaticJobRegistrar registrar() {

    AutomaticJobRegistrar registrar = new AutomaticJobRegistrar();
    registrar.setJobLoader(jobLoader());
    registrar.setApplicationContextFactories(applicationContextFactories());
    registrar.afterPropertiesSet();
    return registrar;

}
```

The registrar has two mandatory properties, one is an array of ApplicationContextFactory (here created from a convenient factory bean), and the other is a JobLoader. The JobLoader is responsible for managing the lifecycle of the child contexts and registering jobs in the JobRegistry.

The ApplicationContextFactory is responsible for creating the child context and the most common usage would be as above using a ClassPathXmlApplicationContextFactory. One of the features of this factory is that by default it copies some of the configuration down from the parent context to the child. So for instance you don't have to re-define the PropertyPlaceholderConfigurer or AOP configuration in the child, if it should be the same as the parent.

The AutomaticJobRegistrar can be used in conjunction with a JobRegistryBeanPostProcessor if desired (as long as the DefaultJobLoader is used as well). For instance this might be desirable if there are jobs defined in the main parent context as well as in the child locations.

1.6.3. JobOperator

As previously discussed, the `JobRepository` provides CRUD operations on the meta-data, and the `JobExplorer` provides read-only operations on the meta-data. However, those operations are most useful when used together to perform common monitoring tasks such as stopping, restarting, or summarizing a Job, as is commonly done by batch operators. Spring Batch provides these types of operations via the `JobOperator` interface:

```
public interface JobOperator {

    List<Long> getExecutions(long instanceId) throws NoSuchJobInstanceException;

    List<Long> getJobInstances(String jobName, int start, int count)
          throws NoSuchJobException;

    Set<Long> getRunningExecutions(String jobName) throws NoSuchJobException;

    String getParameters(long executionId) throws NoSuchJobExecutionException;

    Long start(String jobName, String parameters)
          throws NoSuchJobException, JobInstanceAlreadyExistsException;

    Long restart(long executionId)
          throws JobInstanceAlreadyCompleteException, NoSuchJobExecutionException,
                  NoSuchJobException, JobRestartException;

    Long startNextInstance(String jobName)
          throws NoSuchJobException, JobParametersNotFoundException,
JobRestartException,
                 JobExecutionAlreadyRunningException,
JobInstanceAlreadyCompleteException;

    boolean stop(long executionId)
          throws NoSuchJobExecutionException, JobExecutionNotRunningException;

    String getSummary(long executionId) throws NoSuchJobExecutionException;

    Map<Long, String> getStepExecutionSummaries(long executionId)
          throws NoSuchJobExecutionException;

    Set<String> getJobNames();

}
```

The above operations represent methods from many different interfaces, such as `JobLauncher`, `JobRepository`, `JobExplorer`, and `JobRegistry`. For this reason, the provided implementation of `JobOperator`, `SimpleJobOperator`, has many dependencies:

```xml
<bean id="jobOperator" class="org.spr...SimpleJobOperator">
    <property name="jobExplorer">
        <bean class="org.spr...JobExplorerFactoryBean">
            <property name="dataSource" ref="dataSource" />
        </bean>
    </property>
    <property name="jobRepository" ref="jobRepository" />
    <property name="jobRegistry" ref="jobRegistry" />
    <property name="jobLauncher" ref="jobLauncher" />
</bean>
```

```java
 /**
  * All injected dependencies for this bean are provided by the @EnableBatchProcessing
  * infrastructure out of the box.
  */
 @Bean
 public SimpleJobOperator jobOperator(JobExplorer jobExplorer,
                                 JobRepository jobRepository,
                                 JobRegistry jobRegistry) {

    SimpleJobOperator jobOperator = new SimpleJobOperator();

    jobOperator.setJobExplorer(jobExplorer);
    jobOperator.setJobRepository(jobRepository);
    jobOperator.setJobRegistry(jobRegistry);
    jobOperator.setJobLauncher(jobLauncher);

    return jobOperator;
 }
```

If you set the table prefix on the job repository, don’t forget to set it on the job explorer as well.

1.6.4. JobParametersIncrementer

Most of the methods on `JobOperator` are self-explanatory, and more detailed explanations can be found on the javadoc of the interface. However, the `startNextInstance` method is worth noting. This method will always start a new instance of a Job. This can be extremely useful if there are serious issues in a `JobExecution` and the Job needs to be started over again from the beginning. Unlike `JobLauncher` though, which requires a new `JobParameters` object that will trigger a new `JobInstance` if the parameters are different from any previous set of parameters, the `startNextInstance` method will use the `JobParametersIncrementer` tied to the `Job` to force the `Job` to a new instance:

```
public interface JobParametersIncrementer {

    JobParameters getNext(JobParameters parameters);

}
```

The contract of JobParametersIncrementer is that, given a JobParameters object, it will return the 'next' JobParameters object by incrementing any necessary values it may contain. This strategy is useful because the framework has no way of knowing what changes to the JobParameters make it the 'next' instance. For example, if the only value in JobParameters is a date, and the next instance should be created, should that value be incremented by one day? Or one week (if the job is weekly for instance)? The same can be said for any numerical values that help to identify the Job, as shown below:

```
public class SampleIncrementer implements JobParametersIncrementer {

    public JobParameters getNext(JobParameters parameters) {
        if (parameters==null || parameters.isEmpty()) {
            return new JobParametersBuilder().addLong("run.id", 1L).toJobParameters();
        }
        long id = parameters.getLong("run.id",1L) + 1;
        return new JobParametersBuilder().addLong("run.id", id).toJobParameters();
    }
}
```

In this example, the value with a key of 'run.id' is used to discriminate between JobInstances. If the JobParameters passed in is null, it can be assumed that the Job has never been run before and thus its initial state can be returned. However, if not, the old value is obtained, incremented by one, and returned.

An incrementer can be associated with Job via the 'incrementer' attribute in the namespace:

```
<job id="footballJob" incrementer="sampleIncrementer">
    ...
</job>
```

The java config builders also provide facilities for the configuration of an incrementer:

```
@Bean
public Job footballJob() {
    return this.jobBuilderFactory.get("footballJob")
                     .incrementer(sampleIncrementer())
                     ...
                     .build();
}
```

1.6.5. Stopping a Job

One of the most common use cases of `JobOperator` is gracefully stopping a Job:

```
Set<Long> executions = jobOperator.getRunningExecutions("sampleJob");
jobOperator.stop(executions.iterator().next());
```

The shutdown is not immediate, since there is no way to force immediate shutdown, especially if the execution is currently in developer code that the framework has no control over, such as a business service. However, as soon as control is returned back to the framework, it will set the status of the current `StepExecution` to `BatchStatus.STOPPED`, save it, then do the same for the `JobExecution` before finishing.

1.6.6. Aborting a Job

A job execution which is `FAILED` can be restarted (if the `Job` is restartable). A job execution whose status is `ABANDONED` will not be restarted by the framework. The `ABANDONED` status is also used in step executions to mark them as skippable in a restarted job execution: if a job is executing and encounters a step that has been marked `ABANDONED` in the previous failed job execution, it will move on to the next step (as determined by the job flow definition and the step execution exit status).

If the process died (`"kill -9"` or server failure) the job is, of course, not running, but the `JobRepository` has no way of knowing because no-one told it before the process died. You have to tell it manually that you know that the execution either failed or should be considered aborted (change its status to `FAILED` or `ABANDONED`) - it's a business decision and there is no way to automate it. Only change the status to `FAILED` if it is not restartable, or if you know the restart data is valid. There is a utility in Spring Batch Admin `JobService` to abort a job execution.