

1. Spring Batch Admin User Guide

Version : 2.0.0.M1

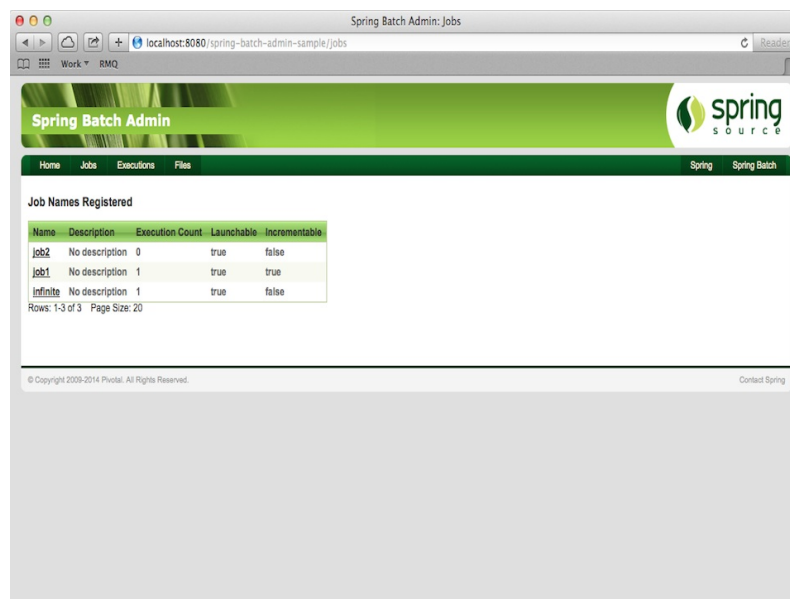
Authors : Dave Syer

Spring Batch Admin provides a web-based user interface that features an admin console for [Spring Batch](#) applications and systems. It is an open-source project from [Spring](#) (Apache 2.0 Licensed).

1.1. Main Use Cases

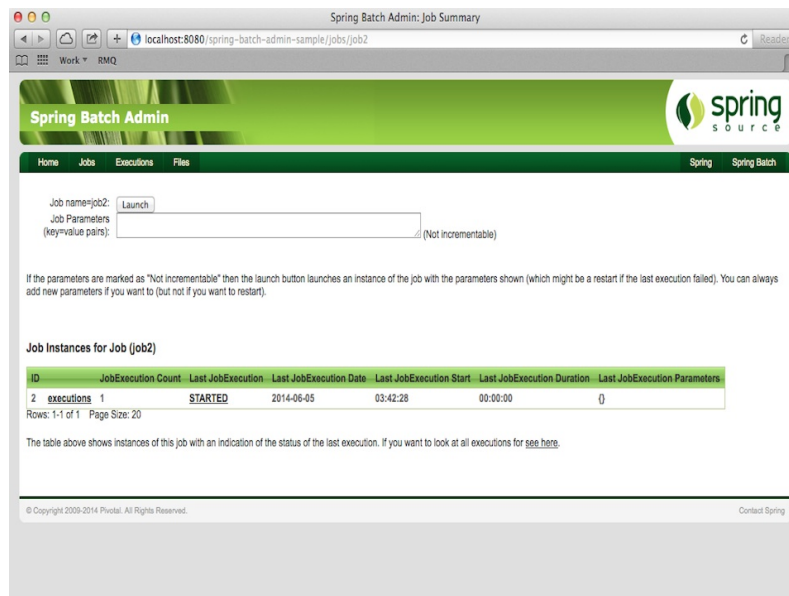
The easiest way to get a quick overview of Spring Batch Admin is to see some screenshots of the main use cases. The user interface is a web application (built with Spring MVC).

1.2. Inspect Jobs



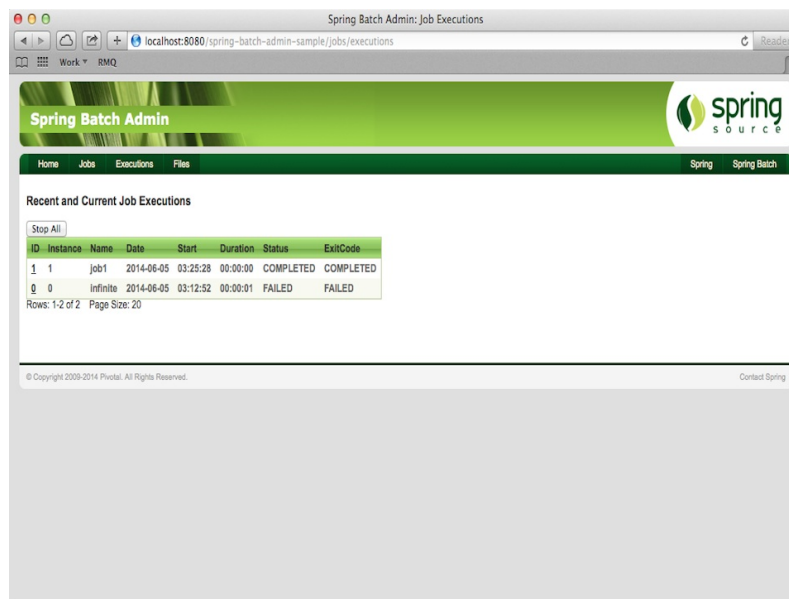
The user can inspect the jobs that are known to the system. Jobs are either launchable or non-launchable (in the screenshot they are all launchable). The distinction is that a launchable job is defined and configured in the application itself, whereas a non-launchable job is detected as state left by the execution of a job in another process. (Spring Batch uses a relational database to track the state of jobs and steps, so historic executions can be queried to show the non-launchable jobs.)

1.3. Launch Job



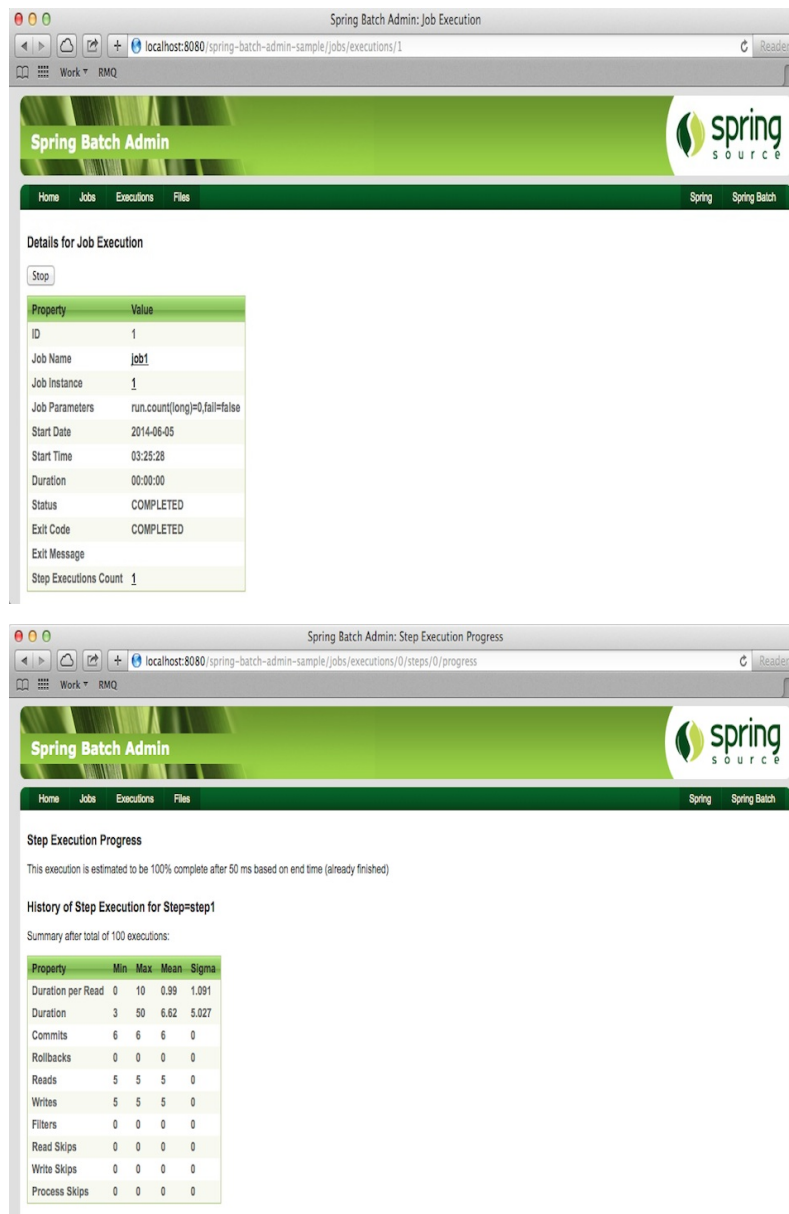
Launchable jobs can be launched from the user interface with job parameters provided as name value pairs, or by an incrementer configured into the application.

1.4. Inspect Executions



Once a job is executing, or has executed, this view can be used to see the most recent executions, and a brief summary of their status (STARTED, COMPLETED, FAILED, etc.).

Each individual execution has a more detailed view (shown above), and from there the user can click down to a view of each of the step executions in the job (only one in this case). A common reason for wanting to do this is to see the cause of a failure.



The top of the step execution detail view shows the history of the execution of this step across all job executions. This is useful for getting a statistical feel for performance characteristics. A developer running a job in an integration test environment might use the statistics here to compare different parameterisations of a job, to see what effect is of changing (for instance) the commit interval in an item processing step.

The bottom of the step execution view has the detailed meta-data for the step (status, read count, write count, commit count, etc.) as well as an extract of the stack trace from any exception that caused a failure of the step (as in the example shown above).

1.5. Stop an Execution

Spring Batch Admin: Job Execution

localhost:8080/spring-batch-admin-sample/jobs/executions/0

Spring Batch Admin

Home Jobs Executions Files

Spring Spring Batch

Details for Job Execution

Abandon

Restart

Property	Value
ID	0
Job Name	infinite
Job Instance	0
Job Parameters	fail=false
Start Date	2014-06-05
Start Time	03:12:52
Duration	00:00:01
Status	FAILED
Exit Code	FAILED
Exit Message	org.springframework.batch.core.JobExecutionException: Flow execution ended unexpectedly at org.springframework.batch.core.job.flow.FlowJob.doExecute(FlowJob.java:141) at org.springframework.batch.core.job.AbstractJob.execute(AbstractJob.java:301) at org.springframework.batch.core.launch.support.SimpleJobLauncher\$1.run(SimpleJobLauncher.java:134) at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142) at java.util.concurrent.ThreadPoolExecutor\$Worker.run(ThreadPoolExecutor.java:617) at java.lang.Thread.run(Thread.java:745) Caused by: org.springframework.batch.core.job.flow.FlowExecutionException: Ended flow-infinite at state-infinite.step1 with exception at org.springframework.batch.core.job.flow.support.SimpleFlow.resume(SimpleFlow.java:160) at org.springframework.batch.core.job.flow.support.SimpleFlow.start(SimpleFlow.java:130) at org.springframework.batch.core.job.flow.FlowJob.doExecute(FlowJob.java:135) ... 5 more Caused by: org.springframework.batch.core.StartLimitExceededException: Maximum start limit exceeded for step: step1StartMax: 100 at org.springframework.batch.core.job.SimpleStepHandler.shouldStart(SimpleStepHandler.java:221) at org.springframework.batch.core.job.SimpleStepHandler.handleStep(SimpleStepHandler.java:119) at org.springframework.batch.core.job.flow.JobFlowExecutor.executeStep(JobFlowExecutor.java:64) at org.springframework.batch.core.job.flow.support.state.StepState.handle(StepState.java:60) at org.springframework.batch.core.job.flow.support.SimpleFlow.resume(SimpleFlow.java:151) ... 7 more

Spring Batch Admin: Job Execution

localhost:8080/spring-batch-admin-sample/jobs/executions/1

Spring Batch Admin

Home Jobs Executions Files

Spring Spring Batch

Details for Job Execution

Stop

Property	Value
ID	1
Job Name	job1
Job Instance	1
Job Parameters	run.count(long)=0,fail=false
Start Date	2014-06-05
Start Time	03:25:28
Duration	00:00:00
Status	COMPLETED
Exit Code	COMPLETED
Exit Message	
Step Executions Count	1

A job that is executing can be stopped by the user (whether or not it is launchable). The stop signal is sent via the database and once detected by Spring Batch in whatever process is running the job, the job is stopped (status moves from STOPPING to STOPPED) and no further processing takes place.

2. Features

2.1. Installing Spring Batch Admin

There is a sample project in the distribution - it's a regular war file project which you can build with Maven (`mvn install` and look in the target directory). The sample can also be created with a couple of clicks in the SpringSource Tool Suite (STS): go to File->New->Spring Template Project , and select the Spring Batch Admin Webapp from the list (if it isn't there upgrade STS). In STS you can right click on the project and Run As->Run On Server choose Tomcat or tc Server, and the app will deploy and open a web browser (e.g. `localhost:8080/spring-batch-admin-sample`). Check out the features for launching jobs and inspecting job executions. If the STS project looks out of date (look at the version numbers in the pom.xml) then you might be able to upgrade by changing the POM, or else you may have to install a nightly update of STS or wait for the next release.

2.1.1. Building your own applications:

It is really easy to build a custom web application with the ability to inspect job execution data from external processes:

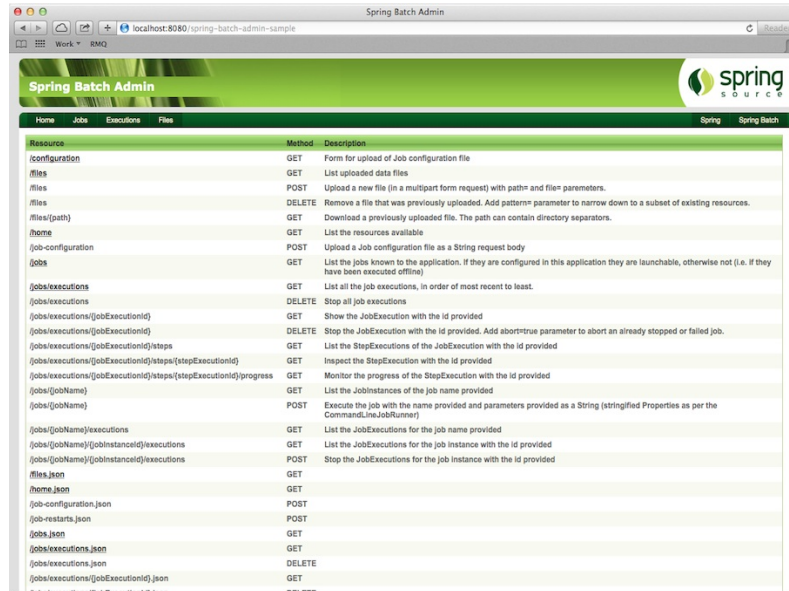
- Create a war project with an `index.jsp` and a `web.xml` (from the sample or from the `spring-batch-admin-resources.jar` in the package `org.springframework.batch.admin.web.resources`).
- Include the `spring-batch-admin-*.jar` files in `WEB-INF/lib` , plus all their dependencies. In the sample this is done simply by making the WAR depend on those jar files in the Maven pom.
- Deploy the web app. It starts up with an in-memory database, so without any jobs defined in the application it isn't going to do much at first. See the section on environment settings for more detail.

See later in this guide for instructions on how to modify the application and add you own jobs for launching.

2.2. Running Spring Batch Admin

If you are deploying Spring Batch Admin as a WAR file (either the sample or one you rolled yourself), just deploy it as normal to a servlet container like [Pivotal tcServer](#) .

If the WAR file is called `spring-batch-admin-sample.war` (it may have a version number appended), then when you load the application in a browser at <http://localhost:8080/spring-batch-admin-sample>) it should show a listing of the contents of the application:



This page is a list of URLs, or URL patterns, along with the HTTP method (GET, POST, etc.) that you need to use to carry out an operation on that resource. If the URL is a pattern it has braces ({} in it to represent the value of a parameter that needs to be supplied by the user. Users of Spring REST support in MVC will recognise this convention, and also the idea that a GET represents a request for information, and a POST is an "insert" operation. So, for example, to request a list of all running Job executions you must GET the resource at /jobs/executions (relative to the batch/ servlet in the application). And to launch a new job execution you POST to /jobs/ jobName (with "{jobName}" replaced with the name of a launchable job).

Links on the home page that are not parameterised and require a GET method are clickable in the browser and take you to that page in the application. The others are all navigable from there.

2.3. Menus

The basic menus for the application out of the box are

- Home - the URL list or site map listing the interface for the application
- Jobs - list and launch jobs. Only jobs which are either configured in the application or have been executed in other processes against the same database show up here.
- Executions - view and interact with running Job and Step executions.
 - Files - upload and list input files and configuration files for jobs.

You can learn how to extend the content of the application including the menus in a later section of this guide.

2.4. JSON API

Spring Batch Admin can act as a JSON web service. All the main HTML UI features can be accessed by JSON clients. In general, the recipe for doing this is to take a normal HTML URL

from the UI application and add a .json suffix. All the JSON endpoints are listed on the home page of the sample application.

The examples below all use curl from the command line, but the same resources could be used with an Ajax request from a browser, or with Spring's RestTemplate (see the integration tests in the project source code for examples of the latter).

2.4.1. Read-only Access using HTTP GET

After deploying the sample application you can list the jobs registered by sending it a GET request:

```
$ curl -s http://localhost:8080/spring-batch-admin-sample/batch/jobs.json
{"jobs" : {
  "resource" : "http://localhost:8080/spring-batch-admin-sample/batch/jobs.json",
  "registrations" : {
    "infinite" : {
      "name" : "infinite",
      "resource" : "http://localhost:8080/spring-batch-admin-sample/batch/jobs/infinite.json",
      "description" : "No description",
      "executionCount" : 0,
      "launchable" : true,
      "incrementable" : false
    },
    "job1" : {
      "name" : "job1",
      ...
    },
    "job2" : {
      "name" : "job2",
      ...
    }
  }
}
```

As you can see from the example above, the information listed for JSON clients is the same as for HTML clients. A JSON response always contains a link to the resource that it came from, and also links to other interesting resources, e.g.

```
$ curl -s http://localhost:8080/spring-batch-admin-sample/batch/jobs/job1.json
{"job" : {
```

```
"resource" : "http://localhost:8080/spring-batch-admin-sample/batch/jobs/job1.json",
"name" : "job1",
"jobInstances" : {
  "0" : {
    "resource" : "http://localhost:8080/spring-batch-admin-
sample/batch/jobs/job1/0/executions.json",
    "executionCount" : 1,
    "lastJobExecution" : "/spring-batch-admin-sample/batch/jobs/executions/0.json",
    "lastJobExecutionStatus" : "FAILED",
    "jobParameters" : {
      "run.count(long)" : "0"
    }
  }
}
}
```


The result here is a list of all the job instances for the job with name "job1". Each instance is identified as a JSON key whose value is the job instance id ("0" in the example above), and contains a link to another resource that you can visit (GET) to drill down into the job executions.

2.4.2. Read-Write Access: Launching a Job

You can launch a job with a POST request to the job resource:

```
$ curl -d jobParameters=fail=false http://localhost:8080/spring-batch-admin-  
sample/batch/jobs/job1.json  
{"jobExecution" : {  
  "resource" : "http://localhost:8080/spring-batch-admin-  
sample/batch/jobs/executions/2.json",  
  "id" : "2",  
  "status" : "STARTING",  
  "startTime" : "",  
  "duration" : "",  
  "exitCode" : "UNKNOWN",  
  "exitDescription" : "",  
  "jobInstance" : { "resource" : "http://localhost:8080/spring-batch-admin-  
sample/batch/jobs/job1/1.json" },  
  "stepExecutions" : {  
  }  
}  
}
```

The input is a set of JobParameters specified as a request parameter called "jobParameters". The format of the job parameters is identical to that in the UI (a comma or new-line separated list of name=value pairs). The output is a JSON object representing the JobExecution that is running. In the example, there are no step executions yet because they haven't started running. The resource link on line 2 can be used to inspect the result and poll for updates to the status and the step executions. E.g. (with another GET):

```
$ curl -s http://localhost:8080/spring-batch-admin-sample/batch/jobs/executions/2.json  
{"jobExecution" : {  
  "resource" : "http://localhost:8080/spring-batch-admin-  
sample/batch/jobs/executions/2.json",  
  "id" : "3",  
  "status" : "COMPLETED",  
  "startTime" : "17:44:52",  
  "duration" : "00:00:00",
```

```
"exitCode" : "COMPLETED",
"exitDescription" : "",
"jobInstance" : { "resource" : "http://localhost:8080/spring-batch-admin-
sample/batch/jobs/job1/2.json" },
"stepExecutions" : {
  "step1" : {
    "status" : "COMPLETED",
    "exitCode" : "COMPLETED",
    "id" : "3",
    "resource" : "http://localhost:8080/spring-batch-admin-
sample/batch/jobs/executions/2/steps/2.json",
    "readCount" : "5",
    "writeCount" : "5",
    "commitCount" : "6",
    "rollbackCount" : "0",
    "duration" : "00:00:00"
  }
}
}
```

2.5. Configuration Upload

An interesting problem that has no universal good solution is: how do I change the configuration of a running Spring application? Spring Batch Admin has a configuration upload feature that solves this problem in a particular way.

2.5.1. Screenshots of the Basic Use Case

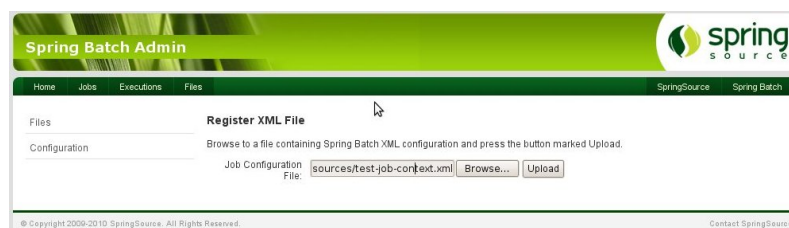


The screenshot shows the 'Jobs' view in the Spring Batch Admin application. It features a navigation bar with 'Home', 'Jobs', 'Executions', and 'Files'. Below the navigation bar, the title 'Job Names Registered' is displayed. A table lists the registered jobs with columns for Name, Description, Execution Count, Launchable, and Incrementable. The table contains three rows: 'infinite', 'job1', and 'job2'. Below the table, it indicates 'Rows: 1-3 of 3' and 'Page Size: 20'. At the bottom, there is a copyright notice: '© Copyright 2009-2010 SpringSource. All Rights Reserved.'

Name	Description	Execution Count	Launchable	Incrementable
infinite	No description	0	true	true
job1	No description	0	true	true
job2	No description	0	true	false

We start with a look at the Jobs view in the application. It shows a list of jobs that are either launchable or monitorable by the web application.

Now the plan is to upload a new Job configuration and see this view change. So we start with the "Files" menu in the top navigation, and then click on "Configuration". This presents a simple form to upload a file. If we point to a Spring XML configuration file, the view looks like this:



The screenshot shows the 'Configuration' form in the Spring Batch Admin application. The form is titled 'Register XML File' and includes a description: 'Browse to a file containing Spring Batch XML configuration and press the button marked Upload.' Below the description, there is a text input field for 'Job Configuration File' containing the path 'sources/test-job-context.xml', followed by 'Browse...' and 'Upload' buttons. The form is part of a larger page with a navigation bar and a footer with copyright information.

We press the "Upload" button and the configuration is uploaded and parsed and the Job instances in the uploaded file are registered for launching:

You can see a new entry in the job registry ("test-job") which is launchable in-process because the application has a reference to the Job . (Jobs which are not launchable were executed out of process, but used the same database for its JobRepository , so they show up with their executions in the UI.)

2.5.2. Variant of the Basic Use Case

Name	Description	Execution Count	Launchable	Incrementable
infinite	No description	0	true	true
job1	No description	0	true	true
job2	No description	0	true	false
test-job	No description	0	true	false

Rows: 1-4 of 4 Page Size: 20

© Copyright 2009-2010 SpringSource. All Rights Reserved.

A common variant of the basic upload use case is to upload a modification of an existing Job configuration, instead of a brand new one. This is common because one often needs to tweak the runtime parameters, especially when performance testing. For instance we could change the `href="http://static.springsource.org/spring-batch/trunk/reference/html/configureStep.html#commitInterval"` to `commit-interval /a` in one of the Steps in a Job so that more (or less) items are grouped in a chunk and committed together during a repetitive item-processing Step . This might affect performance and the Job might run faster overall because of more efficient use of transactional resources. We would be able to measure and verify this directly by launching the Job from the application and inspecting the result (because the web interface is resource oriented, this could also be automated using a simple shell script based on curl).

2.5.3. How Does it Work?

The secret to the implementation of this use case is that Job configuration files are parsed up into separate Spring ApplicationContext instances, which are tracked by their file name. The execution environment (JobLauncher , transaction manager, DataSource etc.) are provided by the application as a parent context, and the Jobs all live in child contexts of that environment. This means that they can be volatile, and their lifecycle can be managed by a service in the application. The service in question happens to be activated by Spring Integration (as described elsewhere in this guide) via a service activator wrapping a JobLoader . JobLoader is a standard component from Spring Batch which tracks and manages ApplicationContext instances containing Jobs .

2.6. Monitoring and Management with JMX

Spring Batch Admin provides some nice features for monitoring and managing Batch jobs remotely using JMX. In addition the standard configuration has JMX MBeans enabled for the Spring Integration components within the application context.

2.6.1. Exposing Metrics for Job and Step Executions

All the basic JMX features are exposed in one place in the Manager jar, in META-INF/spring/batch/bootstrap/manager/jmx-context.xml . The Job and Step metrics come from one bean, which in turn registers a set of MBeans for the Job and Step metrics:

```
<bean id="batchMBeanExporter"
class="org.springframework.batch.admin.jmx.BatchMBeanExporter">
  <property name="server" ref="mbeanServer" />
  <property name="jobService">
    <bean class="org.springframework.aop.framework.ProxyFactoryBean">
      <property name="target" ref="jobService" />
    </bean>
  </property>
  <property name="defaultDomain" value="spring.application" />
</bean>
```

If you start the sample application with the usual command line options to expose the MBeanServer, e.g.

```
-Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.port=1099 -
Dcom.sun.management.jmxremote.authenticate=false -
Dcom.sun.management.jmxremote.ssl.need.client.auth=false
```

then launch one of the jobs, and look at the MBeans in JConsole or your favourite JMX client, you will see the BatchMBeanExporter . As soon as you inspect its attributes, you will also see Job and Step Executions as well (they are lazily registered only when the basic attributes are first inspected):

The BatchMBeanExporter needs a reference to the JobService because it uses the JobRepository to extract most of its metrics. It also has a basic cache for the metric data, to prevent it having to go back to the repository every time it is polled. The default cache settings have an expiry timeout of 60 seconds (so new data will show up in JMX every 60 seconds). The cache can be re-configured by overriding the bean definition for the cacheInterceptor bean.

2.6.1.1. Job Execution Metrics

Once a Job has executed at least once and the BatchMBeanExporter attributes inspected (so the Job execution leaves an imprint on the JobRepository) its metrics will be available to JMX clients. The MBean for these metrics has an ObjectName in the form

Name	Value
ExecutionCount	2
FailureCount	1
JobRunning	false
LatestDuration	84.0
LatestEndTime	2010-12-29 10:13:38.665
LatestExecutionId	1
LatestExitCode	COMPLETED
LatestStartTime	2010-12-29 10:13:38.581
LatestStatus	COMPLETED
LatestStepExitDescription	
MaxDuration	470.0
MeanDuration	277.0

```
spring.application:type=JobExecution,name=job1
```

where job1 is the job name.

Metrics include information about the latest execution (status, duration, exit description, etc.) and also some basic summary information (number of executions, number of failures etc.).

2.6.1.2. Step Execution Metrics

Once a step has executed at least once (and left an imprint on the JobRepository) its metrics will be available to JMX clients. The MBean for these metrics has an ObjectName in the form

```
spring.application:type=JobExecution,name=job1,step=step1
```

where job1 is the parent job name, and step1>> is the step name. Because of the form of the <<<ObjectName , most JMX clients present step execution metrics as a child of the job they belong to, making them easy to locate.

Metrics include information about the latest execution (status, duration, commit count, read and write counts, etc.) and also some basic summary information (number of executions, number of failures etc.).

2.6.1.3. Step Execution Timeouts and Monitoring

The metrics above can be used by JMX clients to monitor the performance of an application and send alerts if service levels are not being achieved. Many JMX clients can do this natively with no additional configuration in the Batch application.

In addition, a convenient JMX MonitorMBean is provided in Spring Batch Admin to allow users to receive notifications if a Step execution is overrunning. To use this just add a bean definition to your Job configuration context, and refer to the Step you want to monitor e.g.

```
<bean id="j1.step1.monitor"
class="org.springframework.batch.admin.jmx.StepExecutionServiceLevelMonitor">
  <property name="jobName" value="job1"/>
  <property name="stepName" value="step1"/>
  <property name="upperThreshold" value="30000"/>
  <property name="defaultDomain" value="spring.application"/>
</bean>
```

This will create an MBean with ObjectName in the form:

```
spring.application:type=GaugeMonitor,name=job1.step1.monitor
```

which can be exposed through the MBeanServer using the standard Spring context shortcut:

```
<context:mbean-export default-domain="spring.application" server="mbeanServer"/>
```

When a step is taking too long to execute, a JMX notification is sent automatically by the MBeanServer to any clients that have registered for notifications. The notification comes with an embedded event in the form:

```
javax.management.monitor.MonitorNotification
[source=spring.application:name=job1.step1.monitor,type=GaugeMonitor]
[type=jmx.monitor.gauge.high]
[message=]
```

where job1.step1.monitor is the name of the Spring bean definition used to define the monitor.

2.6.2. Exposing Spring Integration Metrics

If the Spring Integration features of Spring Batch Admin are being used (file polling, job launching through a MessageChannel etc.), then it might also be useful to look at the standard JMX metrics for the channels and handlers. You can see the MBeans in the screenshot above.

3. Customization

3.1. Environment Settings

The most likely thing you will want to customize is the location of the database. Spring Batch Admin ships with an embedded HSQLDB database, which is initialized on start up.

- To change the database type add a file to the application classpath called `batch-[type].properties`, where `[type]` is the database type you want to use (e.g. `mysql`, `oracle`, `db2`). Copy the contents of the `batch-hsql.properties` from the Manager jar and change the values to suit your environment. Then launch the application with a system property - `DENVIRONMENT=[type]`.
- To stop the database from being wiped and re-created on start up just set `batch.data.source.init=false` (in the properties file or as a System property).

N.B. The use of the environment variable to switch on the database type is only a suggestion. You can change that and other things by overriding and adding configuration fragments to the Spring application context. See below for details.

3.2. Overriding Components from Spring Batch Admin

The system tries to provide some useful defaults for things like transaction manager, job repository, job registry etc. Most of these live in the manager jar in a special place: `META-INF/spring/batch/bootstrap`. If you want to override them, just add your own versions of the same bean definitions to a Spring XML config file in `META-INF/spring/batch/override` (these are guaranteed to load after the bootstrap files, so they can override default definitions). You could use this to override the data source definition as an alternative to the environment settings described above.

3.2.1. Important Bootstrap Components

The important bootstrap components are listed below along with some common scenarios for when you might want to override or replace them:

3.2.1.1. JDBC Data Source

- Bean ID : dataSource
- Default : DBCP BasicDataSource with placeholders for common properties.
- Override Scenarios : Change to another implementation, or add additional placeholders.

3.2.1.2. Spring Transaction Manager

- Bean ID : transactionManager
- Default : DataSourceTransactionManager injected with the dataSource
- Override Scenarios : Change to another implementation. For example if you are using Hibernate in your jobs, you will need to provide a HibernateTransactionManager .

3.2.1.3. Job Launcher

- Bean ID : jobLauncher
- Default : SimpleJobLauncher injected with the jobRepository and transactionManager , and also a thread pool task executor with bean id poolTaskExecutor , so that launches happen in a background thread by default.
- Override Scenarios : Should be fine as it is for most use cases. To switch to synchronous launching change the task executor (see below).

3.2.1.4. Job Launcher Task Executor

- Bean ID : jobLauncherTaskExecutor
- Default : a task executor with pool-size=6 and rejection policy ABORT .
- Description : Injected into the jobLauncher to control the number of cocurrent jobs executing in-process. The effect of the defaults is that at most 6 jobs will run concurrently when launched locally from this application, and any submitted in excess of that will fail.
- Override Scenarios :
 - Change to a synchronous task executor for integration testing (see the sample unit tests for an example), e.g.

```
<bean id="jobLauncherTaskExecutor"  
class="org.springframework.core.task.SyncTaskExecutor"/>
```

- Change the pool parameters. Note that these settings can be changed at runtime using JMX, if the task executor is exposed explicitly.

3.2.1.5. Throttled Task Executor for Jobs and Steps

- Bean ID : throttledTaskExecutor
- Default : a task executor with pool-size=600 and rejection policy CALLER_RUNS and infinite throttle limit.
- Description : Not used anywhere in the manager, but provided as a convenience for applications that provide their own jobs and steps with concurrent behaviour.
- Override Scenarios : The sample contains one job with a concurrent step ("job2"), and it works by creating a local taskExecutor bean inheriting from the root definition, and adding a throttle limit:

```
<bean id="taskExecutor" parent="throttledTaskExecutor">
  <property name="throttleLimit" value="100"/>
</bean>
```

This task executor is then injected into a step in the sample which has the effect of limiting the concurrency of the step to 100, but within overall limits set globally by the bootstrap context, and without discarding or failing any tasks that are submitted. The throttle limit is set to 100 so that if this job is the only one running it will hardly be affected by the throttle, but if it is contending with other jobs, then they will all eventually run in the thread pool provided by the bootstrap (with 600 threads by default). The throttled executor itself delegates to a pool which can be modified independently by overriding its bean definition poolTaskExecutor , e.g.

```
<task:executor id="poolTaskExecutor" pool-size="200" rejection-
policy="CALLER_RUNS"/>
```

changes the pool size to 200 (from 600), limiting all jobs that use it to at most 200 threads. Your mileage may vary with these thread pool settings depending on the workload that is executed. It is worth experimenting with the parameters to see if you can affect the overall throughput or execution times of your multi-threaded jobs.

3.2.1.6. Job Repository

- Bean ID : jobRepository
- Default : a JDBC version of SimpleJobRepository .
- Override Scenarios : Not common to override. The most likely scenario is changing the table prefix or large string column sizes (e.g. for a 2-byte character encoding the maxVarCharLength property would be set to 1/2 the length of the long columns in the database).

3.2.1.7. Job Loader

- Bean ID : jobLoader
- Default : a AutomaticJobRegistrar with path locations set to classpath*/META-INF/spring/batch/jobs/*.xml .
- Override Scenarios :
 - Changing the path locations maybe (trivial but not that useful).
 - Change the inheritance strategy for post processors. By default the AOP configuration and the PropertyPlaceholderConfigurer instances from the root context are copied into the child context created from the Job configuration files. You can change this behaviour just by providing your own versions in addition in the child context, or to make global changes you can override the jobLoader with one that copies different post processors down into the child, or one that does something completely different.
 - Override it to a dummy bean for the purposes of an integration test, so that it doesn't try to create multiple instances of the Job that you want to test. The sample application has an example. In JobExecutionTests-context.xml we want to load the Jobs explicitly into the test context, and prevent them from being loaded again by the job loader, so we do this:

```

<!-- prevent loading of other jobs by overriding the loader in the main bootstrap
context -->
<bean id="jobLoader" class="java.lang.String"/>

```

3.2.1.8. Job Service

- Bean ID : jobService
- Default : a JDBC version of SimpleJobService .
- Override Scenarios : Not common to override. The job service is configured through a factory bean a bit like the JobRepository, and it has the same basic properties (e.g. you can change the table prefix or long column length).

3.3. Add your Own Job Executions

To add job executions from another process just execute a job (e.g. from command line) against the same database that is used by Spring Batch Admin. The UI picks up the meta data from the usual Spring Batch tables.

3.3.1. Create Your Own Standalone Application

To create a standalone application and run jobs against the Spring Batch Admin database, the easiest thing to do is include the spring-batch-admin-*.jar files in the classpath of your application

and use the same conventions to override and extend as you would for a web application.

If you do that then in particular for a standalone application that runs a single job in its own process, you will want to override the `jobLauncherTaskExecutor` as described above, and possibly also the `jobLoader` in addition to providing data source properties. Here's an example configuration file to drive a job:

```
<beans ...>

  <import resource="classpath*:/META-INF/spring/batch/bootstrap/manager/*.xml" />
  <import resource="classpath*:/META-INF/spring/batch/jobs/*.xml" />

  <!-- ensure job runs in foreground thread -->
  <bean id="jobLauncherTaskExecutor"
class="org.springframework.core.task.SyncTaskExecutor"/>
  <!-- prevent loading of other jobs by overriding the loader in the main bootstrap context -->
  <bean id="jobLoader" class="java.lang.String"/>

</beans>
```

With the job defined in `/META-INF/spring/batch/jobs/*.xml` you can use the `CommandLineJobRunner` from Spring Batch to load the file above, and launch the job by name, e.g.

```
$ java org.springframework.batch.core.launch.support.CommandLineJobRunner \
-next config/launch-context.xml jobName
```

3.4. Add your Own Jobs For Launching in the UI

Include Spring XML files in `META-INF/spring/batch/jobs`. Each file should be self-contained (except for Batch execution components like the `jobRepository` which are provided centrally), and carry one or more bean definitions for a Spring Batch Job. When the application starts these files are scanned and loaded as child contexts, and then the jobs are registered with a `JobRegistry` in the parent. Because they are child contexts you don't have to worry about name clashes between different XML files or different contributing JARs (except for the jobs which must have unique names).

As a convenience, the child contexts inherit property placeholders and AOP configuration from the parent (this is not the default behaviour for a child context in Spring). This means you can control those things centrally if you need to. Of course, the child can always create its own placeholder definition and AOP configuration, but these will not affect the parent or any of its siblings.

3.4.1. Create your own Web Application

Hints for custom applications:

- Look at the sample.
- Create a war project with an index.jsp and a web.xml (from the sample or from the spring-batch-admin-resources.jar).
- Include the spring-batch-admin-*.jar files in WEB-INF/lib . In the sample this is done simply by making the WAR depend on those jar files in the Maven pom.
- Optionally add your own jobs in the classpath under META-INF/spring/batch/jobs/*.xml .
- Deploy the web app.

3.5. Design and Architecture

Spring Batch Admin is a layered, extensible application. Its main artifacts are 2 JAR files (libraries) containing all the content and business logic for the web application. To deploy them you need a deployment platform, like a WAR file in a servlet container. There is a sample application spring-batch-admin-sample which show how this works in practice.

3.6. Application Context Structure

There is a root or parent application context, and a child context for the UI components. These are both loaded in the standard Spring MVC way following directives in web.xml. There are also possibly several additional child contexts (of the main root) containing job configurations.

- The root context is loaded from a file in the manager JAR, but all that does is import from well known locations on the classpath:

```
<import resource="classpath*/META-INF/spring/batch/bootstrap/**/*.xml"/>
<import resource="classpath*/META-INF/spring/batch/override/**/*.xml"/>
```

Note that the "override" location has no files in it in Spring Batch Admin distribution. This is a placeholder for users to add their own content.

- The UI child context is loaded in the same way from

```
<import resource="classpath*/META-INF/spring/batch/servlet/**/*.xml"/>
```

N.B. the child context by default only loads when you visit the first page in the application. You should be able to see which files it actually loads from the log statements that come out on the server console.

- The job contexts are loaded by a special component (ClasspathXmlJobLoader from Spring Batch) in the parent context. It looks for individual files in the pattern classpath*/META-INF/spring/batch/jobs/*.xml and loads each file individually as its own self-contained context, registering any instance of Job it finds in the JobRegistry at the top level.

The job contexts are loaded in this way so that you can provide your own jobs in multiple (possibly many) JAR files included in the application, but you do not have to worry too much about duplicate bean definitions because only the Job names have to be unique. A job context inherits the AOP and PropertyPlaceholderConfigurer settings from the root context (not the standard behaviour of a child context) as a convenience, but you can add your own settings to apply locally as well.

3.7. Extending the UI

There are three extension points for the UI: changing or translating text, adding menus, and modifying the content of the existing pages.

3.7.1. URL Paths

URL paths start at the context root "/" by default. If you want to use it with a different context path, just change the servlet mapping in web.xml and override the resourceService bean in the root application context to change the servlet path.

3.7.2. Text Content

Text content that doesn't come from the JobRepository is rendered using a standard Spring MessageSource . To change it you need to provide a bean in your servlet application context (META-INF/spring/batch/servlet/override/*.xml) with id "messageSource" of type MessageSource . E.g.

```
<bean id="messageSource"
class="org.springframework.context.support.ResourceBundleMessageSource">
  <property name="basename" value="messages"/>
</bean>
```

loads a resource bundle (as per java.util.Resource) from classpath:messages.properties . See the Spring MVC documentation for more detail on how message sources work.

The message source is a set of key-value pairs where the key is provided by the application and the value is what you want to be rendered. To find all the possible key values you would have to

inspect all the Freemarker templates in the resources and manager jars, and also the view bean definitions (where for example the page titles are defined) and controller definitions (for error codes).

(A complete list of message keys would be a great addition to the project. If anyone wants to automate the compilation of that list please feel free to make a contribution.)

3.7.3. Adding a Menu

Create a component of type `org.springframework.batch.admin.web.base.Menu` in the UI child context (`META-INF/spring/batch/servlet/override/*.xml`). The Menu has a `url` property that should point to a controller mapping (relative to the application root context). That's it really.

If you also want the page you add to fit in with the information architecture of the application as a whole, i.e. show the other menus and links, you will want to implement the view for the menu that you add in a specific way. That is: you will use a Freemarker template to render the body of the page, and refer to it in a Spring MVC View definition in your servlet XML configuration. The template can use the standard layout in the application as a parent bean, and that way it inherits the information architecture (menus and styling). Look in the manager jar for `META-INF/spring/batch/servlet/manager/manager-context.xml` where you will find some examples. They look like this:

```
<bean name="jobs" parent="standard">
  <property name="attributes">
    <props merge="true">
      <prop key="body">/manager/jobs/jobs.ftl</prop>
      <prop key="titleCode">jobs.title</prop>
      <prop key="titleText">Spring Batch Admin: Jobs</prop>
    </props>
  </property>
</bean>
```

The only mandatory property in this is the "body", which is the location of a Freemarker template for the main body (business content) of the page. Freemarker templates by default have paths relative to either `classpath:/org/springframework/batch/admin/web` or `/WEB-INF/web` (either works but the `WEB-INF` location is dynamically reloadable at development time). The other properties shown above are to override the page title: the "code" is a key in the Spring `MessageSource` and the "text" is the default value if the code cannot be resolved.

3.7.4. Modifying Existing Pages

3.7.4.1. Styles and Branding

To change the colors, fonts, branding and layout of existing pages, you can use a cascading style sheet. The default style sheets are in `spring-batch-admin-resources.jar` under `META-INF/resources/styles`. Most of the things you might need to change will be in `main.css` or `local.css`. To override the settings here you could put a modified copy of those files in your WAR under `/styles` at the top level of your applications (the Spring JS ResourceServlet is used to search the context and all jars in that order). Or you could modify the standard view bean or its template (see below) to import a style sheet from a different location.

3.7.4.2. View Bean Overrides

In the UI each view is a Spring bean, so one way to change the default views is to override those bean definitions in `META-INF/spring/batch/servlet/override/*.xml`. The existing menu bars and static content are defined in `spring-batch-admin-resources.jar`, while the dynamic and Batch-specific content are defined in `spring-batch-admin-manager.jar`. In both cases there are view beans defined in `META-INF/spring/batch/servlet/**/*.*.xml`. Some views have html content only and some have json content, in which case the view name ends with `.json`, e.g. in the manager jar `META-INF/spring/batch/servlet/manager/manager-context.xml` we have the definitions for the view that renders the list of jobs:

```
<bean name="jobs" parent="standard">
  <property name="attributes">
    <props merge="true">
      <prop key="body">/manager/jobs/html/jobs.ftl</prop>
      <prop key="titleCode">jobs.title</prop>
      <prop key="titleText">Spring Batch Admin: Jobs</prop>
    </props>
  </property>
</bean>

<bean name="jobs.json" parent="standard.json">
  <property name="attributes">
    <props merge="true">
      <prop key="body">/manager/jobs/json/jobs.ftl</prop>
    </props>
  </property>
</bean>
```

To override one or other of these you would need to create a Spring config file in `META-INF/spring/batch/servlet/override/*.xml` and copy the definitions above replacing any of the properties as desired.

3.7.4.3. Template Overrides

Another way to modify the existing views is to override just the Freemarker templates.

In the JARs the templates are located in sub-directories of `org.springframework.batch.admin.web`, and you can override them by putting files with the same name and relative path in `WEB-INF/web`. For example, the whole look and feel would change if you added a file `WEB-INF/web/layouts/html/standard.ftl` to your application.

Files in the `WEB-INF/web` location are also reloadable at development time.