Summer Batch - Reference Guide

The Summer Batch team <Summer.Batch.Team@bluage.com>

Summer Batch - Reference Guide

by The Summer Batch team

1.1.2 RELEASE - July 2016

Copyright © 2015-2016 Bluage Corporation, All Rights Reserved.

Copies of this document may be made for your own use and for distribution to others, provided that you do not charge any fee for such copies and further provided that each copy contains this Copyright Notice, whether distributed in print or electronically.

Table of Contents

1. Introduction 1
    - Summer Batch Introduction 1
    - Typical use cases 1
    - Architecture overview 2
    - Programming guidelines 2
    - Version History 3
        - version 1.1.2 - July 2016 3
        - version 1.1.1 - June 2016 3
        - version 1.1.0 - March 2016 4
        - version 1.0.0 - November 2015 4
2. Supported Features 5
    - The JSR 352 as starting point 5
    - Main features 5
    - Additional Features 5
3. Summer batch vocabulary and mechanisms 7
    - Summer batch vocabulary 7
    - Job 7
    - Step 7
    - Batchlet 7
    - Chunk handling 7
        - Item Reader 8
        - Item Processor 8
        - Item Writer 8
    - Repository 8
    - Operator and Batch Runtime 8
    - The batch mechanisms 9
        - Typical chunk oriented step behavior 9
        - The tasklet approach 10
4. Configuration 12
    - Job Specification Language (JSL) 12
        - XSL Configuration 12
        - Job Configuration 12
        - Step Element 13
        - Flow Element 14
        - Split Element 15
        - Batch Control Flow 15
    - Unity Configuration 17
        - Batch Configuration 17
        - Batch Artifacts 18
        - Additions to Unity 19
5. Running a Job 23
    - Several ways to run a job 23
    - Using a JobStarter 23
        - JobStarter Methods. 24
        - Fine grained control over job executions. 24
6. Using Basic Features 25
    - Reading and writing flat files 25
        - Using a flat file reader 25
        - Using a flat file writer 28
    - Reading from and writing to RDBMS 32
        - Reading from a database 32
        - Writing to a database 35
        - Database Support 37
7. Using Advanced Features 40
    - Reading and writing EBCDIC files using COBOL copybooks 40

Using the `EbcdicFileReader` 40
Using the `EbcdicFileWriter` 44
FTP operations support 45
FTP put operations 45
FTP get operations 46
Email sending support 47
Empty file check support 49
Sql Script Runner support 49
Sort Tasklet 50
Supported DFSORT Features 51
Generation Data Groups (GDG) 52
Context Managers 53
`ContextManager` class 53
`ContextManagerUnityLoader` 55
`AbstractExecutionListener` and `AbstractService` 55
Process Adapters 55
Template facility 56
Controlling Template ID 57
A. Appendix 59
Job File Format Xml schema 59
Ebcdic File Format Xml schema 60
Database Repository scripts per vendor 61
Sql Server scripts 61
Oracle scripts 62
IBM DB2 scripts 64
NOT OFFICIALLY SUPPORTED : PostgreSQL scripts 65
Index 67

List of Figures

1.1. Summer Batch dependencies 2
3.1. Chunk oriented step: basic transactional behavior 9
3.2. Chunk oriented step: when transaction roll-back occurs 10
7.1. EBCDIC File Format XML schema 41

List of Examples

3.1. XML Job Configuration 7
3.2. XML Job Configuration 8
4.1. XSL declaration configuration: 12
4.2. A Non Restartable Job 12
4.3. Job Configuration 13
4.4. XML Chunk Specification 13
4.5. XML Batchlet Specification 13
4.6. XML step configuration with listener 14
4.7. XML Partition Specification 14
4.8. XML Flow Specification 15
4.9. XML Split Specification 15
4.10. XML Two step Specification 15
4.11. XML conditional step Specification 16
4.12. XML conditional step Specification 16
4.13. Setting Database Persistence of Job Explorer and Job Repository 17
4.14. Creating a New Registration 21
4.15. Registration of a List of Character Strings 21
5.1. Sample entry point method using `JobStarter` 23
6.1. `FlatFileItemReader` declaration in the job XML file 26
6.2. Sample delimited flat file data 26
6.3. Sample flat file target business object 26
6.4. Sample flat file target business object field set mapper 27
6.5. Delimited flat file reader - sample unity configuration 27
6.6. `FlatFileItemWriter` declaration in the job XML file 29
6.7. Sample flat file writer input business object 30
6.8. Formatted flat file writer - sample unity configuration 31
6.9. Query parameters supported syntax examples 32
6.10. `DataReaderItemReader` target business object 33
6.11. `DataReaderItemReader` target business object RowMapper 34
6.12. `DataReaderItemReader` declaration in the job XML file 34
6.13. Formatted flat file writer - sample unity configuration 34
6.14. `DatabaseBatchItemWriter` target business object 36
6.15. `DatabaseBatchItemWriter` declaration in the job XML file 37
6.16. Database batch writer - sample unity configuration 37
6.17. Adding support for other RDBMS : the PostgreSQL example 38
7.1. Sample XML copybook export 40
7.2. Sample business object to which EBCDIC records will be mapped 41
7.3. Sample EBCDIC reader mapper 42
7.4. `EbcdicFileReader` declaration in the job XML file 43
7.5. `EbcdicFileReader` Unity configuration 43
7.6. `EbcdicFileWriter` declaration in the job XML file 44
7.7. `EbcdicFileWriter` Unity configuration 44
7.8. `FtpPutTasklet` usage in the job XML file 46
7.9. `FtpPutTasklet` Unity configuration 46
7.10. `FtpGetTasklet` usage in the job XML file 47
7.11. `FtpGetTasklet` Unity configuration 47
7.12. `EmailTasklet` usage in the job XML file 48
7.13. `EmailTasklet` Unity configuration 48
7.14. Typical `EmptyFileCheckTasklet` usage in the job XML file 49
7.15. Sample Unity configuration for a `EmptyFileCheckTasklet` 49
7.16. Typical `SqlScriptRunnerTasklet` usage in the job XML file 50
7.17. Sample Unity configuration for a `SqlScriptRunnerTasklet` 50
7.18. Registration of the GDG Resource Loader 52
7.19. GDG Configuration 53
7.20. `IContextManager` interface contract 54

7.21. Unity wiring example for `ContextManager` 55
7.22. Example ProcessAdapter Wiring 56
7.23. Example format file 56
7.24. More advanced example format file 56
7.25. Typical `AbstractTemplateLineAggregator` usage 57
7.26. Typical TemplateLineAggregator class 57
7.27. TemplateLineAggregator unity setup with a ProcessAdapter 57
7.28. TemplateLineAggregator usage in a process 58
7.29. `GetParameters` method for heterogeneous inputs 58

Chapter 1. Introduction

Summer Batch Introduction

Summer Batch is an efficient, open-source, lightweight batch framework for .NET community taking full advantages of the C# platform and aimed at fulfilling typical enterprise bulk processing needs. Well known Java Batch Standard JSR-352 is used as guideline for building framework and features of JSR-352 are supported by Summer Batch.

Our primary goals are to help users to:

- build new lightweight and efficient batch solution for Microsoft®-based environments.
- provide modern architecture patterns that facilitates migrating batch to modern Microsoft®-based environments;

This is the result of a several months collaboration between Accenture and BluAge® Corporation. Accenture has a long experience in dealing with enterprise batch frameworks, on a large set of platforms. BluAge® Corporation is a leading actor in providing legacy modernization solutions. By sharing our experiences, we managed to focus on very essentials parts of JSR-352 specification, covering needs for majority of users, in particular in scope of legacy batch modernization.

As a result of this partnership, BluAge® has developed two software solutions:

- a product to modernize COBOL batches to a Summer Batch solution;
- a product to craft automatically a Summer Batch solution from an UML2 model.

Typical use cases

- To treat large sets of data, with a typical read-process-write repetitive scenario
- Bulk processing has to be non-interactive, highly efficient and scalable
- C# your favorite development and runtime platform

Summer Batch provides following notable features (non-exhaustive list -- see chapter 2 for details) :

- Restartability of failed jobs using a database persisted job repository;
- Sequential or parallel processing of jobs (scalability support);
- Mainframe EBCDIC files readers and writers, using COBOL copybooks;
- File sort capabilities using legacy DFSORT cards semantics;
- FTP operations support;
- Email sending support;
- SQL Scripts invocation support;
- Mainframe GDG-like facility support;

To achieve this, Summer Batch relies on following bricks :

- Microsoft® .NET framework 4.5, using C# as development language;
- Unity 3.5 as Dependency Injection container (see Unity MSDN page);
- NLog 4.1.2 (see NLog project home page);

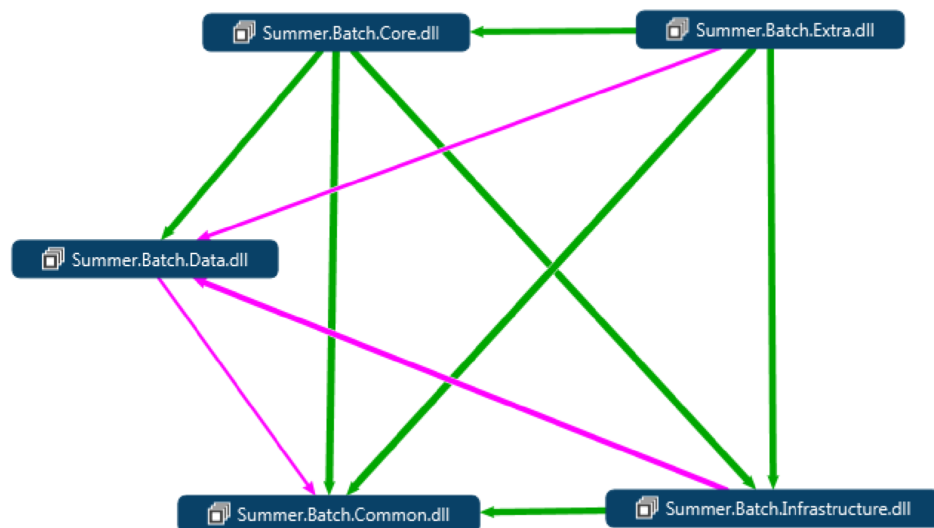
Architecture overview

Summer Batch solution is split into 5 components:

- Summer Batch Common : Shared artifacts, used by other components (Collections, I/O, Proxies, etc.);
- Summer Batch Infrastructure : Basic item readers and writers, and repeat support elements;
- Summer Batch Data: Database access dedicated artifacts;
- Summer Batch Core : Main component that contains batch engine, job repository, launchers, listeners, etc ...;
- Summer Batch Extra : Additional readers/writers (EBCDIC), dedicated batchlets implementations (FTP support, Email support, ...) and additional framework facilities;

Following schema exposes inter-dependencies between them (an arrow between two components means that component A uses component B : A → B).

Figure 1.1. Summer Batch dependencies



Programming guidelines

Key objectives of batch programming are: reliability, performance and maintainability.

Performance is dependent on a large set of factors, but some guidelines should be kept in mind :

- Shorten the path between data and data processor, Reducing network overhead can have a significant positive impact on global batch performance when dealing with databases.
- For critical performance needs, using parallel processing facility, which requires some extra work to be done (smart data partitioning).
- Carefully specify chunk size :
  - A small chunk size will lower global batch performance by adding a lot of checkpoints operations;
  - A very large chunk size will increase global batch performance but
    - could have a significant negative impact on memory consumption;
    - could involve some serious perturbations if data are being consumed by other appliances, in case of a batch failure (as rollback will deal with a large set of data).
- Hunt for unnecessary operations in code; In particular, be careful with logger calls that can degrade overall performance when used on a granularity basis.
- Follow general SQL statement performance guidelines. Some examples:
  - Make sure SQL queries are performing well, and that all databases indexes have been created;
  - Pay attention to the order of filtering (WHERE clause) by inspecting execution plan, to ensure the most discriminant filters are invoked first.
  - Hunt and remove unnecessary outer joins.
  - Only select fields that are really needed by processing.
- Keep I/O consumption profile low;

Some additional suggestions for maintainability and reliability:
Keep it simple. Have some intricate steps sequences within a given job can make code hard to maintain and harder to test.

Important

This reference guide makes assumption that you know Visual Studio and creating projects and solutions. However, to help users set up their first Summer Batch project, we wrote Getting Started guide. Please read this tutorial prior to starting this reference documentation.

Version History

version 1.1.2 - July 2016

This version is a minor bug-fix release of SummerBatch. List of fixes bugs:

- Issue with TransactionScopeManager;
- Issue with disposable instances in the Step Scope;

version 1.1.1 - June 2016

This version is a minor bug-fix release of SummerBatch. List of fixes bugs:

- Issue with FileUtilsTasklet;

version 1.1.0 - March 2016

The following new features have been included in version 1.1.0 :

- Support for files comparison in FileUtilsTasklet;
- Support for PowerShell scripts invocation;
- Support for multiple outputs in the sort tasklet.

version 1.0.0 - November 2015

This version is the first G.A. version of SummerBatch.

Chapter 2. Supported Features

The JSR 352 as starting point

Summer Batch uses main architecture carried by JSR-352 specification. JSR-352 is standard specification used in the java-world for batch-oriented software. It supplies a common vocabulary, key concepts and abstractions, that java batch framework should follow, and was designed by main specialists of batch processing. While implementation is in java, concepts are universal.

In addition, while Summer Batch supplies most functionalities that any Batch framework also supplies (such as reading a database), it also contains a set of more specialized features, particularly some useful tools to adapt COBOL batches to Microsoft® .NET world.

Main features

Key features of Summer Batch are as listed below. Explore advance features of Summer Batch in additional feature section.

- Repeatable and customizable batch jobs
- Multi step jobs, with simple step sequences or conditional logic between them
- In-memory or persisted job repository
- Support for a Read-Process-Write logic, as well as arbitrary batchlet steps for a more complete control on behavior
- Chunk-processed steps, with checkpoint management and restartability
- Step partitioning used for parallel processing
- Database readers and writers, with support for Microsoft® SQL Server, IMB® DB2 and Oracle® databases
- Flat file readers and writers
- Easy mapping between readers and writers and your domain classes
- Batch contexts at step level and job level
- XML design for main batch architecture, C# design for step properties

Additional Features

These features are more advanced. Some of them may be specific to Summer Batch technology

- Support for steps without reader or without writer
- Support for steps with several writers
- Support for conditionally executed writers
- Support for key-grouping of read data, to process each group as a whole
- Support for reader and writer use during a process run (useful for multi read)

- SQL script execution steps
- Email sending steps
- FTP get and put steps
- Common file operations, such as copying or deleting files during a job
- File sort capabilities using legacy DFSORT cards semantics
- Mainframe EBCDIC file readers and writers, using COBOL copybooks
- Advanced report writers, either through flat file templates or through Microsoft® Reporting
- Mainframe GDG-like support

Chapter 3. Summer batch vocabulary and mechanisms

Summer batch vocabulary

The batch main concepts, as described and standardized by JSR-352, are quite commonly known and used. Thus, they are reused as familiar items in the Summer Batch framework. A typical Summer Batch program will contain "Jobs", "Steps", "Readers" and "Writers", with out of box C# implementations and a great extensibility that allows users to supply their own.

Batch developers ought to understand vocabulary explained below, as it describes these key concepts and is reused throughout this documentation.

Job

A job represents a batch as a whole. It contains one or several steps that must run together as a flow. It can be launched, it can succeed or fail, and it may be restarted on failure. It is described in an XML document which specifies its name and its step flow.

Example 3.1. XML Job Configuration

```xml
<job id="PayorReport">
  <step id="CleanDatabase" next="GenerateReport">
    <batchlet ref="CleanDatabaseBatchlet" />
  </step>
  <step id="GenerateReport">
    <batchlet ref="GenerateReportBatchlet" />
  </step>
</job>
```

Step

Each phase of a job is represented by a step. A step is an atomic part of a job. It can fail, usually failing whole job. It can be an arbitrary task, executing a piece of program and returning a return code, but most of the time it is a Read-Process-Write task.

Batchlet

When a step is a simple task, it is described in a XML document as a simple reference to a class that performs task as a whole. In previous example, job was a sequence of two simple batchlets.

Chunk handling

Most of the time, a step is a read-process-write task, and manipulated data is processed through subsets of a given size. This is called chunk handling. Each so-called chunk will be used as a checkpoint during processing. When a step fails, current chunk will be rollbacked, while all previous processing will be saved. And on restart (if restart was enabled, which

requires a database-persisted repository), the job will be restarted at exact chunk where failure happened.

In following example, step will be processed through groups of 1000 records read by reader.

Example 3.2. XML Job Configuration

```
...
<step id="TitleUpdateStep">
  <chunk item-count="1000">
    <reader ref="TitleUpdateStep/ReadTitles" />
    <processor ref="TitleUpdateStep/Processor" />
    <writer ref="TitleUpdateStep/UpdateTitles" />
  </chunk>
</step>
...
```

Item Reader

The Item reader is step phase that retrieves data from a given source (database, file, etc.). It supplies items from source until no more are available, in which case it will return null, and its processing is complete.

Item Processor

The Item Processor is step phase that processes data retrieved by reader. It can be used for any kind of manipulations: filtering depending on a business logic, field updates and complete transformation into a different kind of element. It will return result of processing, which may be initial element as is, the initial element with updates, or a completely different element. If it returns null, it means read element is ignored, (thus filtering data read from source).

In case no processor is supplied, read data is transmitted as is to writer.

Item Writer

The Item Writer is the final step phase that writes items to a target (database, file, etc.). It processes the elements given by the processor chunk by chunk, enabling rollback mechanics explained above.

Repository

The Job Repository is where the elements stated above are persisted. It can be in memory or database, However it must be in database if restarting feature is used. It will store the jobs and steps, each job instance (meaning each job run, for example when a job is run each week), and each job and step execution (meaning each attempt to run: when a job instance fails, it can be restarted and there will be a new execution of the same instance and a new execution of the failed step). The job and step contexts are also saved for consistency when a job is restarted.

Operator and Batch Runtime

The Job Operator is the summer batch engine. It is used to interact with repository, start or restart a job, query for existing jobs, etc. The Batch Runtime is entry point of any Summer Batch program. It is a static factory returning a fully setup Job Operator with correct job definition. It enables to get an operator that is ready to start the job.

The batch mechanisms

Most of typical batch processing is being made through chunk oriented steps usage, which are implementing a Read-Process-Write repetitive pattern on data.

Typical chunk oriented step behavior

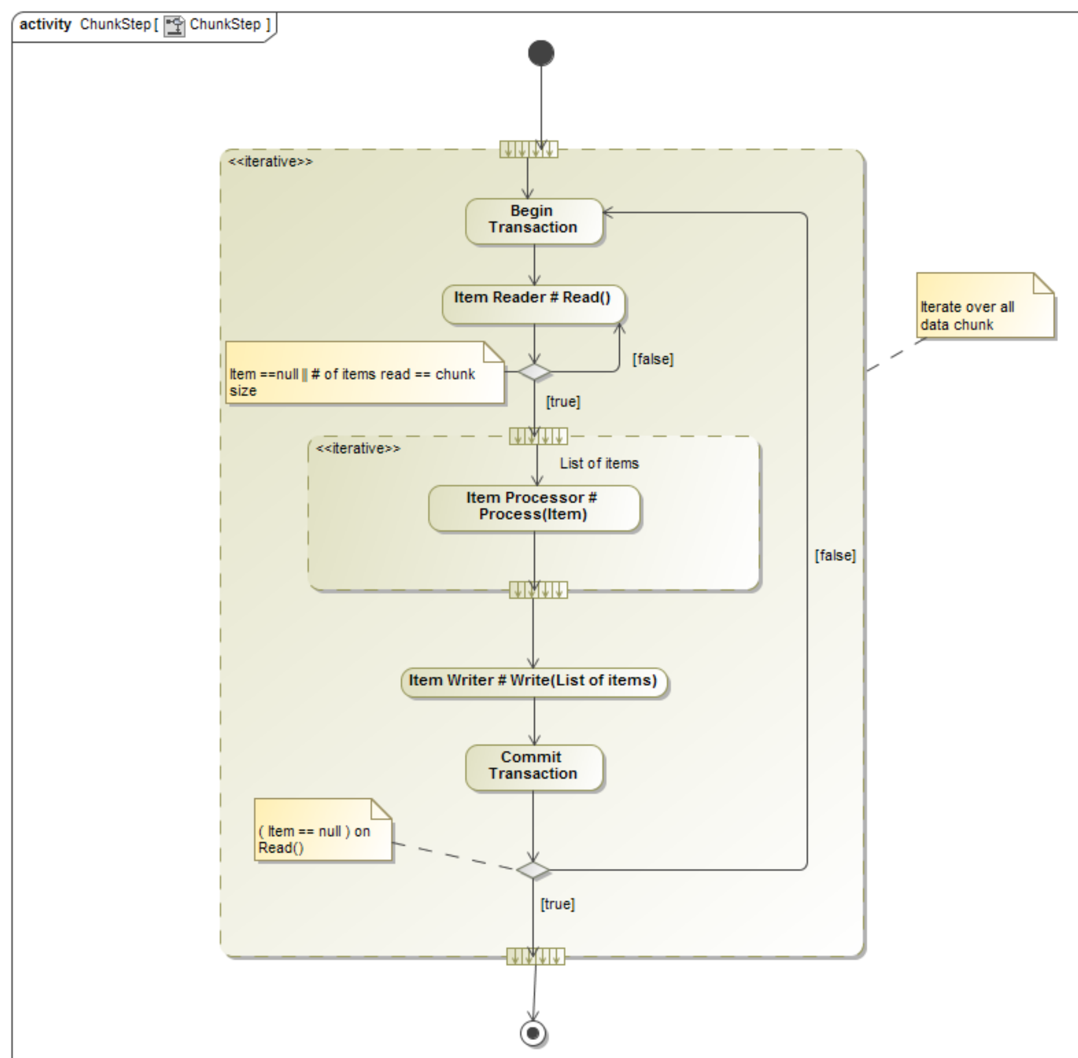
A chunk oriented step is made of:

- An Item Reader
- An Item Processor (Optional)
- An Item Writer

The data to be processed is split into chunks whose size can be optionally defined by using `item-count` attribute (= chunk size);

Each chunk is holding its own transaction. The transactional behavior of chunk oriented step is demonstrated by the figure below:

Figure 3.1. Chunk oriented step: basic transactional behavior



This is basic behavior, when everything runs smoothly and step completes gracefully. The Read-Process-Write pattern is running within transaction boundaries.

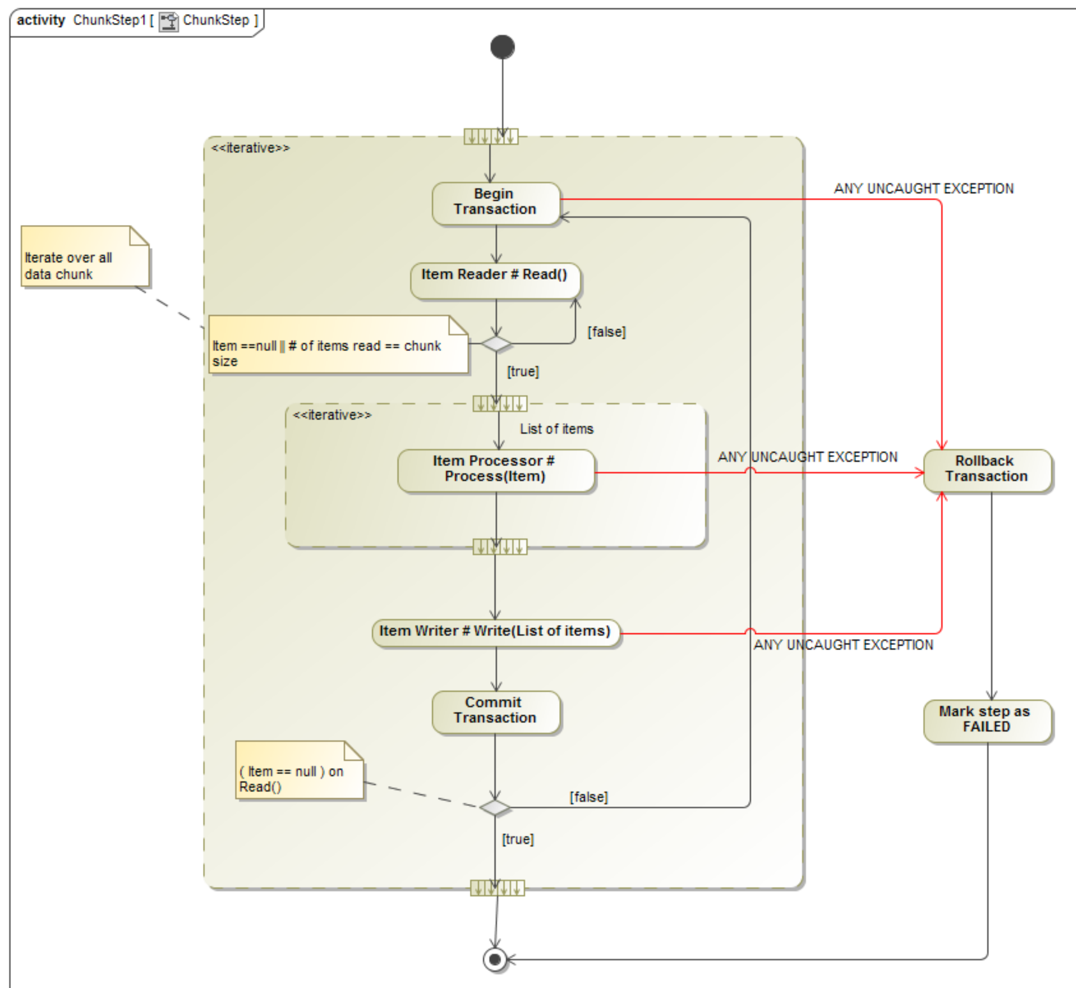
Each time a transaction is committed, job repository is being updated, in order to guarantee a potential job restart (provided repository update is being done on a persistent storage).

Now, what if something goes wrong?

Any uncaught exception thrown during Read-Process-Write operation will lead to rollback of current chunk transaction and will cause the step to FAIL and thus consequently the job to FAIL. If job restartability has been set up, job repository will be updated on transaction rollback so that job can be restarted later.

The figure below illustrates that mechanism

Figure 3.2. Chunk oriented step: when transaction roll-back occurs



Caution

When for any reason we leave step (either because it completed or failed because of an unexpected exception), job repository is updated; this update uses a transaction of its own, clearly separated of any chunk related transaction.

The tasklet approach

Using Chunk oriented step is not the only option; one can use a tasklet (aka a batchlet) to cover a whole step. A batchlet is a class that implements `Summer.Batch.Core.Step.Tasklet.ITasklet` interface.

The transactional support is guaranteed by `Summer.Batch.Core.Step.Tasklet.TaskletStep` class, that is in charge of executing batchlet code (see `DoExecute` method, that delegates to `DoInTransaction` method, that wraps tasklet code effective execution (`Execute` method implementation)).

Chapter 4. Configuration

The job configuration is separated in two: a part describes the content of the job (e.g., steps, flows), while the other defines the artifacts used by the job (e.g., readers, writers) and their fine-grained properties. The former is declared in XML using a subset of the job specification language defined in JSR-352 and the latter is declared in C# using the Unity dependency injection.

Job Specification Language (JSL)

XSL Configuration

The XML Specification language for Summer Batch Configuration can be found at http://www.summerbatch.com/xmlns/SummerBatchXML_1_0.xsd. It can be declared on the specification top element like this:

Example 4.1. XSL declaration configuration:

```
<job id="SampleJob"
    xmlns="http://www.summerbatch.com/xmlns"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.summerbatch.com/xmlns
        http://www.summerbatch.com/xmlns/SummerBatchXML_1_0.xsd">
```

For readability, this XML setup will not be repeated in examples of this section.

Job Configuration

A job is defined using `job` element, which is root element of any JSL file. `id` attribute should contain a unique identifier for the job. A job can contain listeners (with `listeners` element) or batch elements. Listeners are notified when job starts or ends, while batch elements are executable parts of a job.

There are three kinds of batch elements: steps, flows, and splits. Steps are base executables of any job, and are responsible for reading, processing, and reading data. Flows contains other batch elements that are executed using regular flow control as explained in section Batch Control Flow. Splits contains other batch elements that are executed using a task executor (which, for instance, can execute them in parallel, see section Task Executor for more information).

Restartability

If job repository is persisted (see section Job Explorer and Job Repository), job restartability can be set with `restartable` attribute. Default value is `true`.

Example 4.2. A Non Restartable Job

```
<job id="testJob" restartable="false">
        …
</job>
```

Job Listeners

Job listeners can be registered within `listeners` element. Listeners must implement `Summer.Batch.Core.IJobExecutionListener` interface and be registered with the corresponding id in Unity container.

Example 4.3. Job Configuration

```
<job id="testJob">
    <listeners>
        <listener id="testListener" />
    </listeners>
    …
</job>
```

`IJobExecutionListener` interface has two methods, `BeforeJob(JobExecution)` and `AfterJob(JobExecution)` which are called respectively before and after the job execution. `AfterJob` method is called even when job failed; `JobExecution.Status` property exposes job status.

Step Element

A step is defined using `step` element and `id` attribute must be specified. There are two kinds of steps: chunk steps and batchlet steps. Steps can also have listeners and be partitioned.

Chunk Steps

A chunk step has a fixed size and is composed of a reader, a writer, and (optionally) a processor. Step is divided in chunks: records are read until chunk size has been reached, then they are processed, and processed items are finally written. Chunks are repeated until there are no more records to read. Each chunk is executed in a transaction.

`chunk` element has a required attribute, `item-count`—which specifies the chunk size—, and must contain `reader` and `writer` elements. `processor` element is optional.

Example 4.4. XML Chunk Specification

```
<step id="testStep">
    <chunk item-count="10">
        <reader ref="testReader" />
        <processor ref="testProcessor" />
        <writer ref="testProcessor" />
    </chunk>
</step>
```

The reader must implement `IItemReader` interface, the processor must implement `IItemProcessor` interface, and the writer must implement `IItemWriter` interface. All must be registered with the corresponding id in Unity container.

Batchlet Steps

A batchlet is a step that is entirely delegated to a C# class, that must implement `ITasklet` interface. `Execute` method will be executed repeatedly until it returns `RepeatStatus.Finished`. As with chunks, each call is done in a transaction.

Example 4.5. XML Batchlet Specification

```
<step id="testStep">
```

```
    <batchlet ref="testBatchlet" />
</step>
```

Listeners

As with jobs, listeners can be registered with a step using `listeners` element. The listeners must implement `IStepExecutionListener` interface and be registered with the corresponding id in Unity container.

Example 4.6. XML step configuration with listener

```
<step id="testStep">
    <chunk>
        <reader ref="testReader" />
        <processor ref="testProcessor" />
        <writer ref="testProcessor" />
    </chunk>
    <listeners>
        <listener ref="myStepListener"/>
    </listeners>
</step>
```

`IStepExecutionListener` interface has two methods, `BeforeStep(StepExecution)` and `AfterStep(StepExecution)` which are called respectively before and after the step execution. `AfterStep` must return an exit status. This Exit status wraps an Exit code, which can be used for step flow control.

Partitioning

Partitioning allows execution of several instances of a step using a task executor. A partitioner implementing `IPartitioner` interface, creates several step execution contexts, which are used to create different instances of the step. To partition a step, add a `partition` element:

Example 4.7. XML Partition Specification

```
<step id="testStep">
    …
    <partition>
        <mapper ref="testPartitioner" grid-size="10" />
    </partition>
</step>
```

The mapper element specifies Unity id of partitioner with `ref` attribute and grid size (which will be provided to partitioner as a parameter) with `grid-size` attribute. Default grid size is 6.

Note

The way the different step instances are executed depends entirely on task executor. See section Task Executor for more information.

Flow Element

The flow element gathers batch elements together and execute them using standard control flow (see section Batch Control Flow). Elements in a flow can only have transitions to elements in the same flow, but flow will itself transition to an element in its parent.

Example 4.8. XML Flow Specification

```
<flow id="testFlow" next="…">
    <step id="testStep1" next="subFlow">
        …
    </step>
    <flow id="subFlow">
        <step id="testStep2">
            …
        </step>
    </flow>
</flow>
```

Split Element

The split element gathers batch elements together and execute them using a task executor, thus elements in a split do not have specified transitions.

Example 4.9. XML Split Specification

```
<split id="testSplit" next="…">
    <flow id="testFlow">
        …
    </flow>
    <step id="testStep">
        …
    </step>
</split>
```

Note

The way different batch elements are executed depends entirely on task executor. See section Task Executor for more information.

Batch Control Flow

The execution of batch elements in jobs and flows follows a certain control flow. First element is executed and then next element is chose using a set of rules explained in this section.

Straight Control Flow

The simplest control flow executes the elements in a predetermined sequence. This is achieved with `next` attribute, which contains the id of next batch element to execute.

Example 4.10. XML Two step Specification

```
<job id="testJobWithTwoSteps">
    <step id="step1" next="step2">
        …
    </step>
    <step id="step2">
        …
    </step>
```

```
</job>
```

Note

The first batch element to appear in the job will always be the first step to be executed. Other ones may appear in any order, as only `next` attribute is relevant. However, it is advised to write them in correct order.

Conditional Control Flow

The batch element to execute can also depend on status of execution of the previous batch element by using `next` element instead of `next` attribute. `next` element has two required attributes: `on` and `to`. `on` attributes contains a string representing a status and `to` contains the id of a batch element. `next` elements are processed in order: the first that matches status of the previous execution will designate the next element. If no element matches, job fails.

Example 4.11. XML conditional step Specification

```
<job id="testJobWithConditionalSteps">
    <step id="step1">
        …
        <next on="FAILED" to="step2"/>
        <next on="*" to="step3"/>
    </step>
    <step id="step2">
        …
    </step>
    <step id="step3">
        …
    </step>
</job>
```

Note

Here, `FAILED` is standard Summer Batch code for step failure. `*` value in the second `next` element is usual wildcard for *any value*. The status of a step can be customized using `AfterStep` method of step listener.

End and Fail Control flow

In addition to `next` element, `end` or `fail` elements can be used. Both terminates the job, but latter forces a failure.

Example 4.12. XML conditional step Specification

```
<job id="testJobWithEndFailSteps">
    <step id="step1">
        …
        <fail on="SomeFailureCode"/>
        <next on="*" to="step2"/>
    </step>
    <step id="step2">
        …
        <end on="SomeTerminationCode"/>
        <next on"*" to="step3"/>
```

```
    </step>
    <step id="step3">
        …
    </step>
</job>
```

Unity Configuration

Artifacts referenced in the JSL (using `ref` attribute) must be registered in a Unity container using `UnityLoader` class. It has two protected methods that can be overridden: `LoadConfiguration(IUnityContainer)`, which makes registrations for base configuration of the batch, and `LoadArtifacts(IUnityContainer)`, which registers the artifacts of the job (e.g., readers, writers).

Batch Configuration

The implementation of `LoadConfiguration` in `UnityLoader` does required registrations by default, thus it is not required to override it unless this default configuration is needs to be modified for the current batch.

The only mandatory interface to register is `IJobOperator`, but default implementation, `SimpleJobOperator` does require registrations of interfaces `IJobExplorer`, `IJobRepository`, `IJobLauncher`, and `IListableJobLocator`. If using splits or partitioned steps, it is also required to register a task executor (`ITaskExecutor`).

Job Operator

The job operator offers a basic interface to manage job executions (e.g., starting, restarting jobs). It is recommended to use default implementation, `SimpleJobOperator`, but users are free to provide their own implementation.

Job Explorer and Job Repository

The job explorer manages the persistence of batch entities while job repository holds all associated meta-information. Both can either be persisted in a database or stored in memory. Database persistence is required to restart a job but in-memory explorer and repository are useful during development.

For the job repository to be persisted in a database, an ad hoc list of tables must be created in a database schema. Creation (and drop) SQL scripts are being provided for the three supported RDBMS (MS SQL Server, Oracle database and IBM DB2), for details please refer to corresponding appendix section:

- Scripts for MS SQL Server
- Scripts for Oracle database
- Scripts for IBM DB2

When using the default implementation of `UnityLoader.LoadConfiguration`, one can override `PersistenceSupport` property to choose whether to use database or memory explorer and repository. By default, property returns `false`.

Example 4.13. Setting Database Persistence of Job Explorer and Job Repository

```
protected override bool PersistenceSupport { get { return true; } }
```

When using database persistence, an instance of `ConnectionStringSettings` with name “Default” must be registered in Unity container.

Job Launcher

The job launcher is responsible for launching a job. By default, `UnityLoader` uses an instance of `SimpleJobLauncher` with a instance of `SyncTaskExecutor` as a task executor.

Job Locator

The job locator, which implements `IListableJobLocator`, is used to retrieve a job configuration from its name. If it also implements `IJobRegistry`, `UnityLoader` will register in it all jobs that have been registered in Unity container. By default an instance of `MapJobRegistry` is used.

Task Executor

The task executor determines how splits and partitioned steps are executed. Default implementation registered by `UnityLoader`, `SimpleAsyncTaskExecutor`, executes tasks in parallel. Summer Batch also provides `SyncTaskExecutor` which executes tasks synchronously, in sequence.

To have a fine grained control over task execution; users can define their own task executor. A task executor must implement `ITaskExecutor` interface. It has one method, `void Execute(Task)`, which is responsible for executing the task passed as parameter. It is not required that task finishes before `Execute` returns, thus multiple tasks can be executed at the same time.

Batch Artifacts

`UnityLoader.LoadArtifacts` method is responsible for registering all the artifacts required for job in unity container. In particular these artifacts must be registered: readers, processors, writers, tasklets, job listeners, and step listeners.

Readers
: Readers must implement `IItemReader<T>` interface. “`T Read()`” method returns the next read items or `null` if there are more items. Type of returned items `T`, must be a reference type.

Processors
: Processors must implement `IItemProcessor<TIn, TOut>` interface. “`TOut Process(TIn)`” method transforms the item returned by reader (of type `TIn`) to an item of type `TOut` for the writer. If current item must be skipped, then processor should return `null`. Type of returned items `TOut`, must be a reference type.

Writers
: Writers must implement `IItemWriter<T>` interface. “`void Write(IList<T>)`” method writes the items returned by processor or reader (if there are no processor).

Takslets
: Takslets must implement `ITasklet` interface. “`RepeatStatus Execute(StepContribution, ChunkContext)`” method must implement a batchlet step and is repeatedly executed until it returns `RepeatStatus.Finished`.

Job listeners
: Job listeners must implement `IJobExecutionListener` interface. “`void BeforeJob(JobExecution)`” method is called before the job starts and “`void AfterJob(JobExecution)`” method is called after the job ends.

Step listeners
: Step listeners must implement `IStepExecutionListener` interface. “`void BeforeStep(StepExecution)`” method is called before the step

starts and the "ExitStatus AfterStep(StepExecution)" method is called after the step ends. AfterStep method return an exit status that can be used for conditional step control flow in the XML configuration (see section Batch Control Flow)

Scopes

Batch artifacts can be defined in two different *scopes*: *singleton* scope and *step* scope. A scope determines the lifetime of an instance during the execution of batch and is critical when executing steps in parallel or when sharing artifacts between different steps.

Singleton Scope

Artifacts defined in the singleton scope use `Microsoft.Practices.Unity.ContainerControlledLifetimeManager` lifetime manager, which means that only one instance will be created during the whole job execution.

The singleton scope is the default scope. When a resolution is made and no lifetime manager has been defined, `ContainerControlledLifetimeManager` lifetime manager is used.

Step Scope

Artifacts defined in step scope use `Summer.Batch.Core.Unity.StepScope.StepScopeLifetimeManager` lifetime manager. One instance will be created for each step execution where the artifact is used. It means that if a step is executed several times (e.g., with a partitioner), the artifacts defined in step scope (like the reader or the writer) will not be shared by different executions.

Note

When an artifact in singleton scope references an artifact in step scope, a proxy is created. When proxy is used it will retrieve appropriate instance of the artifact, depending on the current step execution. Proxy creation is only possible with interfaces, which means that the reference in the singleton scope artifact must use an interface.

Additions to Unity

To add new features and simplify the syntax, several additions to Unity have been made.

Scope extensions

Two Unity extensions are used to manage scopes (see section Scopes), `Summer.Batch.Core.Unity.Singleton.SingletonExtension` and `Summer.Batch.Core.Unity.StepScope.StepScopeExtension`. Both are automatically added to Unity container by `UnityLoader`. `SingletonExtension` ensures that the default lifetime manager is `ContainerControlledLifetimeManager` instead of `PerResolveLifetimeManager`. `StepScopeExtension` manages references between non step scope artifacts and step scope artifacts, as described in section Step Scope.

Initialization callback

A third extension is added to the container by `UnityLoader`: `Summer.Batch.Core.Unity.PostprocessingUnityExtension` . It is used to execute a callback method once an instance of a class has been initialized. If an instance created by Unity container implements `Summer.Batch.Common.Factory.IInitializationPostOperations` interface, `AfterPropertiesSet` method will be called after the initialization is completed and successful. This

is used to ensure an artifact is correctly configured (e.g., to make sure a reader has been given a resource to read).

New Injection Parameter Values

Unity uses injection parameter values to compute at resolve time value that are injected. Four new types of injection parameter values have been introduced.

`Summer.Batch.Core.Unity.Injection.JobContextValue<T>`	Injects a value from job context. The constructor takes key of the job context value to get as parameter. It value is not of type `T`, it converts the value using its string representation.
`Summer.Batch.Core.Unity.Injection.StepContextValue<T>`	Injects a value from step context. The constructor takes the key of step context value to get as parameter. If value is not of type `T`, it converts the value using its string representation.
`Summer.Batch.Core.Unity.Injection.LateBindingInjectionValue<T>`	Injects a value computed from a string given as a constructor parameter. First it computes a string value from the original string parameter by replacing any late binding sequence (i.e., "#{…}") by a value computed at resolve time. Then the string value is converted into required type. The late binding injection parameter value currently supports three sources for late binding sequence: (1) `jobExecutionContext`, which injects a value from job context using `JobContextValue`, (2) `stepExecutionContext`, which injects a value from step context using `StepContextValue`, and (3) `settings`, which injects a string value read from application settings, in "`App.config`". For each source, a string key must be provided using the following syntax: `#{`*`source`*`['`*`key`*`']}`. For instance, the string `"#{jobExecutionContext['output']}\result.txt"` with `"C:\output"` as the job context value for `"output"` would be replaced by `"C:\output\result.txt"`.
`Summer.Batch.Core.Unity.Injection.ResourceInjectionValue`	Injects a resource. Resources represent files that can be read or written and are used by the readers and writers in Summer Batch. This injection parameter value takes a string an converts it to a file system resource that is injected. This string can contain late binding and is resolved using `LateBindingInjectionValue`. The constructor takes a Boolean optional parameter, `many`. If `many` is true, a list or resources will be returned instead of a single resource. There are two ways to specify several resources in input string: (1) separating different paths with ";", and (2) using ant-style paths with "?", "*", and "**" wildcards.

New Registration Syntax

The `Summer.Batch.Core.Unity.Registration<TFrom, TTo>` class simplifies use of injection members and injection parameter values. An instance of `Registration` represents a registration of a type to a Unity container. Several methods allows configuration of registration, and `Register()` method performs actual registration.

There are two ways to create an instance of `Registration`: to use a constructor or to use one of the extension methods for Unity containers (defined in Summer.Batch.Core.Unity.RegistrationExtension). For instance in Example 4.14, “Creating a New Registration”, `registration1` and `registration2` are equivalent and `registration3` and `registration4` are equivalent.

Example 4.14. Creating a New Registration

```
var registration1 = new Registration<IInterface, ConcreteClass>(
    new ContainerControlledLifetimeManager(), container);
var registration2 = container.SingletonRegistration<IInterface, ConcreteClass>();

var registration3 = new Registration<ConcreteClass, ConcreteClass>(
    "name", new StepScopeLifetimeManager(), container);
var registration4 = container.StepScopeRegistration<ConcreteClass>("name");
```

Configuring a Registration

The `Registration` class has several methods to configure a registration, which all return current registration to allow chained calls. These configuration methods are separated in two types: injection member methods and injection parameter value methods. Injection member methods specify a type of injection (e.g., constructor injection) and injection parameter value methods specify a value to inject. When an injection parameter value method is called, the corresponding parameter value is added to injection member specified by the last injection member method called. In Example 4.15, “Registration of a List of Character Strings” a list of character strings is registered using the `Constructor` injection member method and the `Value` injection parameter value method.

Example 4.15. Registration of a List of Character Strings

```
container.SingletonRegistration<IList<string>, List<string>>("names")
         .Constructor().Value(new []{ "name1", "name2" })
         .Register();
```

Injection Member Methods

`Constructor()`
: Specifies constructor injection. All of the following injection parameter values will be used as parameter for constructor. The actual constructor is inferred from parameter values at resolve time. There can only be one constructor injection per registration.

`Method(string)`
: Specifies method injection. This parameter is the name of the method to call and all of the following injection parameter values will be used as parameter for the method. Actual method is inferred from the name and parameter values at resolve time. There may be several method injections per registration (even for the same method).

`Property(string)`
: Specifies property injection. This parameter is the name of property to set and the following injection parameter value is the set value. There may be several property injections per registration.

Injection Parameter Value Methods

`Instance(object)` or `Value(object)`
: Injects the object passed as parameter. Both methods are identical and the distinction is merely semantical (`Instance` for reference types and `Value` for value types).

`Reference<T>(string)`
: Injects an object that is resolved from Unity container *at resolve time*. Injected object is resolved using "type" passed as generic parameter and "name" passed as pa-

	rameter. The name is optional. Since resolution of injected object is done at resolve time, it does not need to already be registered.
`References<T>(params Type[])`	Injects an array of objects that are resolved from unity container *at resolve time*. The objects are resolved using types passed as parameter.
`References<T>(params string[])`	Injects an array of objects that are resolved from unity container *at resolve time*. The objects are resolved using type passed as generic parameter and names passed as parameter.
`References<T>(params Reference[])`	Injects an array of objects that are resolved from unity container *at resolve time*. The objects are resolved using `Reference` instances passed as parameter. The structure `Summer.Batch.Core.Unity.Reference` contains a type (`Type` property) and a name (`Name` property).
`LateBinding<T>(string)`	Injects an object using the late binding injection parameter value, `LateBindingInjectionValue<T>`. See section New Injection Parameter Values for more information on late binding.
`Resource<T>(string)`	Injects a resource using the resource injection parameter value, `ResourceInjectionValue`. See section New Injection Parameter Values for more information on resource injection.
`Resources<T>(string)`	Injects a list of resources using the resource injection parameter value, `ResourceInjectionValue`. See section New Injection Parameter Values for more information on resource injection.

Chapter 5. Running a Job

Several ways to run a job

Batch solutions need to be operable in multiple environments. Typically, batches executions are often triggered by schedulers or by other scripts languages (DOS, Ant, ...).

Using a JobStarter

The `Summer.Batch.Core.JobStarter` is a facility to help users write their entry point. It contains two public methods: Start and Restart, both taking a job XML file path and a UnityLoader implementation as parameters.

Writing a correct job XML configuration is explained in chapter 4.

The JobStarter can be easily used in a typical Main entry point method :

Example 5.1. Sample entry point method using `JobStarter`

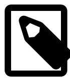
```
using Summer.Batch.Core;
using System;
using System.Diagnostics;

namespace Com.Netfective.Bluage.Batch.Jobs
{
    /// <summary>
    /// Class for launching the restart sample job.
    /// To restart use -restart as the first argument of the Main function.
    /// </summary>
    public static class RestartSampleLauncher
    {
     public static int Main(string[] args)
     {

#if DEBUG
      var stopwatch = new Stopwatch();
      stopwatch.Start();
#endif

      JobExecution jobExecution;
      if (args != null && args.Length >= 1)
      {
        // restarting a job is handled through an argument. Using '-restart' is
        // the only permitted argument for the entry point.
        if (args[0] != "-restart")
        {
          Console.WriteLine("Unknown option :["+args[0]+"]. Only option is -restart");
          return (int) JobStarter.Result.InvalidOption;
        }
        jobExecution = JobStarter.ReStart(@"Batch\Jobs\restart_sample_job.xml",
             new RestartSampleUnityLoader());
      }
      else
      {
        jobExecution = JobStarter.Start(@"Batch\Jobs\restart_sample_job.xml",
             new RestartSampleUnityLoader());
      }
#if DEBUG
  stopwatch.Stop();
     Console.WriteLine(Environment.NewLine + "Done in {0} ms.",
          stopwatch.ElapsedMilliseconds);
     Console.WriteLine("Press a key to end.");
     Console.ReadKey();
#endif
  //returns a integer code value; clients using this Main
  //will be aware of the batch termination result.
```

```
        return (int) (jobExecution.Status == BatchStatus.Completed ?
            JobStarter.Result.Success
            : JobStarter.Result.Failed);
    }
}
```

JobStarter Methods.

Code snippet above displays two typical uses of `JobStarter` class, with `Start` and `Restart` methods. In addition to these, two other methods exist that enable to stop or abandon a current job execution.

- `Start` : Enables to start a new job execution for the specified job;
- `Restart` : Enables to restart a failed job execution. This requires that a failed or stopped job execution exists for the job in parameter. If several such executions exist, last one will be restarted;
- `Stop` : Stops a running job execution for the job in parameter. This requires that such a running execution exists. If several running job executions are found for the specified job, they will all be stopped;
- `Abandon` : Abandons a stopped job execution for the job in parameter. This requires that such a stopped execution exists. If several stopped executions are found, they will all be abandoned. While a stopped job may be restarted, and abandoned one cannot.



Note

`Restart`, `Stop` and `Abandon` all require a persisted job repository to operate.

Fine grained control over job executions.

As seen above, in the `JobStarter` class, `Restart` restarts the last failed or stopped job execution, while `Stop` and `Abandon` operate on all eligible executions. In order to get a fine grained control over job executions and be able to restart, stop or abandon a specific one, key class to use is `Summer.Batch.Core.Launch.BatchRuntime`. Given a job XML file path and IUnityContainer implementation, one can get the corresponding job operator (A class that implements `Summer.Batch.Core.Launch.IJobOperator` interface) using GetJobOperator(UnityLoader loader, XmlJob job) method.

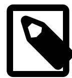
The `IJobOperator` holds all needed operations to get a full control over job executions. In particular, `Restart`, `Stop` and `Abandon` methods all take an *executionId* as a parameter.

- `long? Start(string jobName, string parameters)` : Starts a new job execution with a job name and a list of parameters (comma (or newline) separated 'name=value' pairs string);
- `long? Restart(long executionId)` : Restart a failed job execution corresponding to the provided executionId;
- `bool Stop(long executionId)` : Stops a running job execution corresponding to the provided executionId;
- `JobExecution Abandon(long jobExecutionId)` : Abandons a stopped job execution corresponding to the provided executionId.

Note

In most cases, such a direct use of job operator is not needed.

Chapter 6. Using Basic Features



Preliminary note

For concision, "Summer.Batch." expression may be abbreviated on some occasions using the "S.B." expression, when writing fully qualified names of classes or interfaces;

e.g.:

`Summer.Batch.Infrastructure.Item.File.FlatFileItemReader<T>`
may be abbreviated as

`S.B.Infrastructure.Item.File.FlatFileItemReader<T>`

Reading and writing flat files

Reading and writing flat files is a very common batch; flat files are still largely used to share information between components of the same system or to integrate data coming from outer systems.

Using a flat file reader

A flat file reader implementation is directly available within Summer Batch, and covers typical needs. The class to use is `Summer.Batch.Infrastructure.Item.File.FlatFileItemReader<T>`. The template object T represents business object that will be filled by the records read from flat file. Consequently, some mapping has to be done between the records and their properties of target business object: this is achieved using a line mapper that must be provided at initialization time. A line mapper is a class implementing `Summer.Batch.Infrastructure.Item.File.ILineMapper<out T>` interface.

A line mapper should implement

```
T MapLine(string line, int lineNumber)
```

generic method; this method returns a T object given a line (and its line number within file, which might be relevant).

A default implementation is provided :

`Summer.Batch.Infrastructure.Item.File.Mapping.DefaultLineMapper<T>`

The mapping is done in a two phases process:

- Read line from the flat file is split into fields, using a tokenizer that must be specified at initialization time (class that implements `Summer.Batch.Infrastructure.Item.File.Transform.ILineTokenizer` interface). Two implementations are being provided to cover the most typical needs:
  - `Summer.Batch.Infrastructure.Item.File.Transform.FixedLengthTokenizer`: for lines with a fixed-length format. The fields are being specified using ranges (see `Summer.Batch.Infrastructure.Item.File.Transform.Range`);
  - `Summer.Batch.Infrastructure.Item.File.Transform.DelimitedLineTokenizer`: for lines holding separated fields, e.g. CSV files (the separator string is configurable and defaults to comma).

- The result of first phase is a field set.

Note

see

`Summer.Batch.Infrastructure.Item.File.Transform.IFieldSet` interface and default implementation

`Summer.Batch.Infrastructure.Item.File.Transform.DefaultFieldSet`

Its fields will be mapped to a business object properties using a field set mapper (class that implements

`Summer.Batch.Infrastructure.Item.File.Mapping.IFieldSetMapper` interface).

Field set mappers are bound to your business model; each target business object (intended to be filled by records read from flat file) should have an available mapper.

Now let's see a sample. First, the XML job configuration.

Example 6.1. `FlatFileItemReader` declaration in the job XML file

```
<step id="FlatFileReader">
 <chunk item-count="1000">
  <reader ref="FlatFileReader/FlatFileReader" />
  ...
 </chunk>
</step>
```

Wiring unity configuration is a bit more complex. Our sample makes uses of a semicolon (";") separated flat file whose records will be mapped to the `FlatFileBO` business object. Here is sample flat file data, which we'll be using:

Example 6.2. Sample delimited flat file data

```
1;FlatFile1 ; FlatFile1 ;20100101
2;FlatFile2 ; FlatFile2 ;20100101
```

Example 6.3. Sample flat file target business object

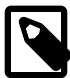
```
using System;

namespace Com.Netfective.Bluage.Business.Batch.Flatfile.Bos
{
    /// <summary>
    /// Entity FlatFileBO.
    /// </summary>
    [Serializable]
    public class FlatFileBO
    {
        /// <summary>
        /// Property Code.
        /// </summary>
        public int? Code { get; set; }

        /// <summary>
        /// Property Name.
        /// </summary>
```

```
        public string Name { get; set; }

        /// <summary>
        /// Property Description.
        /// </summary>
        public string Description { get; set; }

        /// <summary>
        /// Property Date.
        /// </summary>
        public DateTime? Date { get; set; }

    }
}
```

The mapping between data and targeted business object is achieved using `IFieldSetMapper`

Example 6.4. Sample flat file target business object field set mapper

```
using Summer.Batch.Extra;
using Summer.Batch.Infrastructure.Item.File.Mapping;
using Summer.Batch.Infrastructure.Item.File.Transform;

namespace Com.Netfective.Bluage.Business.Batch.Flatfile.Bos.Mappers
{
    /// <summary>
    /// Implementation of <see cref="IFieldSetMapper{T}" /> that creates
    /// instances of <see cref="FlatFileBO" />.
    /// </summary>
    public class FlatFileMapper : IFieldSetMapper<FlatFileBO>
    {
        private IDateParser _dateParser = new DateParser();

        /// <summary>
        /// Parser for date columns.
        /// </summary>
        private IDateParser DateParser { set { _dateParser = value; } }

        /// <summary>
        /// Maps a <see cref="IFieldSet"/> to a <see cref="FlatFileBO" />.
        /// <param name="fieldSet">the field set to map</param>
        /// <returns>the corresponding item</returns>
        /// </summary>
        public FlatFileBO MapFieldSet(IFieldSet fieldSet)
        {
            // Create a new instance of the current mapped object
            return new FlatFileBO
            {
                Code = fieldSet.ReadInt(0),
                Name = fieldSet.ReadRawString(1),
                Description = fieldSet.ReadRawString(2),
                Date = _dateParser.Decode(fieldSet.ReadString(3)),
            };
        }
    }
}
```



Note

Note the use of a dedicated helper (`Summer.Batch.Extra.IDateParser`) to handle DateTime. By default `Summer.Batch.Extra.DateParser` is provided, but you can provide your own implementation to cover more specific needs. Please review API doc to see what services are provided by `DateParser`.

Now that all bricks are set, let's build the unity configuration.

Example 6.5. Delimited flat file reader - sample unity configuration

```
/// <summary>
/// Registers the artifacts required for step FlatFileReader.
/// </summary>
/// <param name="container">the unity container to use for registrations</param>
private void RegisterFlatFileReader(IUnityContainer container)
{
    // Reader - FlatFileReader/FlatFileReader
    container.StepScopeRegistration<IItemReader<FlatFileBO>,
       FlatFileItemReader<FlatFileBO>>("FlatFileReader/FlatFileReader")
        .Property("Resource")
         .Resource("#{settings['BA_FLATFILE_READER.FlatFileReader.FILENAME_IN']}")
        .Property("Encoding").Value(Encoding.GetEncoding("UTF-8"))
        .Property("LineMapper")
         .Reference<ILineMapper<FlatFileBO>>("FlatFileReader/FlatFileReader/LineMapper")
        .Register();

    // Line mapper
    container.StepScopeRegistration<ILineMapper<FlatFileBO>,
       DefaultLineMapper<FlatFileBO>>("FlatFileReader/FlatFileReader/LineMapper")
        .Property("Tokenizer")
         .Reference<ILineTokenizer>("FlatFileReader/FlatFileReader/Tokenizer")
        .Property("FieldSetMapper")
         .Reference<IFieldSetMapper<FlatFileBO>>
          ("FlatFileReader/FlatFileReader/FieldSetMapper")
        .Register();

    // Tokenizer
    container
     .StepScopeRegistration<ILineTokenizer,
      DelimitedLineTokenizer>("FlatFileReader/FlatFileReader/Tokenizer")
        .Property("Delimiter").Value(";")
        .Register();

    // Field set mapper
    container
     .StepScopeRegistration<IFieldSetMapper<FlatFileBO>,
      FlatFileMapper>("FlatFileReader/FlatFileReader/FieldSetMapper")
        .Register();

    // ... -- processor and writer registration is not being shown here --
}
```

Note

- All registrations within unity container are made using Step Scope (`container.StepScopeRegistration`);
- In addition to the mandatory `LineMapper` property, `FlatFileItemReader` uses :
  - A resource to be read (= the flat file); In the sample, resource path is read from the Settings.config file using a key;
  - To specify flat file encoding; Use the `Encoding`.`GetEncoding` static methods family to provide a proper encoding (Optional);
  - Other optional properties not shown in the sample :
    - `LinesToSkip` : given number of lines will be skipped at the start of resource;
    - `Strict` : flag for the strict mode; in strict mode, an exception will be thrown if specified resource does not exist (vs. a simple warn logged in non-strict mode);

Using a flat file writer

Provided implementation is in

`Summer.Batch.Infrastructure.Item.File.FlatFileItemWriter<T>` class, where `T` is the type of business object that will be "dumped" into flat file. `FlatFileItemWriter` uses the following properties:

- Mandatory properties (to be set at initialization time):
  - `Resource` : resource to be written to;
  - `LineAggregator`: a class implementing

    `Summer.Batch.Infrastructure.Item.File.Transform.ILineAggregator<in T>` interface ; this class is responsible for aggregating the business object properties into a single string that can be used to write a line into target flat file.
- Optional properties (some having default values):
  - `LineSeparator` : line separator for the lines to write in flat file; defaults to `System.Environment.NewLine`;
  - `Transactional` : a flag to specify if the writer should take part in the active transaction (meaning that data will be effectively written at commit time); defaults to true;
  - `AutoFlush` : a flag to specify if the writer buffer should be flushed after each write; defaults to false;
  - `SaveState`: a flag to specify if the state of the item writer should be saved in the execution context when Update method is called; defaults to true;
  - `AppendAllowed`: a flag to specify if an existing resource should be written to in append mode; defaults to false;
  - `DeleteIfExists`: a flag to specify if an existing resource should be deleted; if AppendAllowed is set to true, this flag is IGNORED; defaults to false;
  - `DeleteIfEmpty`: a flag to specify if an empty target resource (no lines were written) should be deleted; defaults to false;
  - `HeaderWriter`: a header writer (class implementing the

    `Summer.Batch.Infrastructure.Item.File.IHeaderWriter` interface); Used to write the header of file; No default value;
  - `FooterWriter`: a footer writer (class implementing the

    `Summer.Batch.Infrastructure.Item.File.IFooterWriter` interface); Used to write the footer of file; No default value;

The writing process takes a business object as input, transforms it into a string using `LineAggregator` and append the string to target resource.

Now let's review an example; first, the job XML configuration :

Example 6.6. `FlatFileItemWriter` declaration in the job XML file

```
<step id="step1">
 <chunk item-count="1000">
  ...
```

```
  <writer ref="step1/FlatFileWriter" />
 </chunk>
</step>
```

The crucial part is `LineAggregator`; Summer Batch comes with several `ILineAggregator` implementations :

- `Summer.Batch.Infrastructure.Item.File.Transform.DelimitedLineAggregator<T>`: transforms an object properties into a delimited list of strings. Default delimiter is comma, but can set to any arbitrary string; This is the natural choice to write CSV files;
- `Summer.Batch.Infrastructure.Item.File.Transform.FormatterLineAggregator<T>`: transforms an object properties into a string, using a provided format (the computed string is the result of a call to `string.Format` method, using provided format.);
- `Summer.Batch.Infrastructure.Item.File.Transform.PassThroughLineAggregator<T>`: transforms an object to a string by simply calling `ToString` method of the object;

Our example uses `FormatterLineAggregator`, providing a format string through unity configuration; To fully understand unity configuration that follows, used business object `EmployeeDetailBO` must be shown:

Example 6.7. Sample flat file writer input business object

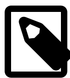
```
using System;

namespace Com.Netfective.Bluage.Business.Batch.Flatfilewriter.Bo
{
    /// <summary>
    /// Entity EmployeeDetailBO.
    /// </summary>
    [Serializable]
    public class EmployeeDetailBO
    {
        /// <summary>
        /// Property EmpId.
        /// </summary>
        public int? EmpId { get; set; }

        /// <summary>
        /// Property EmpName.
        /// </summary>
        public string EmpName { get; set; }

        /// <summary>
        /// Property Name.
        /// </summary>
        public string Name { get; set; }

        /// <summary>
        /// Property EmpDob.
        /// </summary>
        public DateTime? EmpDob { get; set; }

        /// <summary>
        /// Property EmpSalary.
        /// </summary>
        public decimal? EmpSalary { get; set; }

        /// <summary>
        /// Property EmailId.
        /// </summary>
        public string EmailId { get; set; }

        /// <summary>
        /// Property BuildingNo.
        /// </summary>
        public int? BuildingNo { get; set; }
```

```
        /// <summary>
        /// Property StreetName.
        /// </summary>
        public string StreetName { get; set; }

        /// <summary>
        /// Property City.
        /// </summary>
        public string City { get; set; }

        /// <summary>
        /// Property State.
        /// </summary>
        public string State { get; set; }

    }
}
```

Unity configuration:

Example 6.8. Formatted flat file writer - sample unity configuration

```
/// <summary>
/// Registers the artifacts required for step step1.
/// </summary>
/// <param name="container">the unity container to use for registrations</param>
private void RegisterStep1(IUnityContainer container)
{
     //... -- reader and processor registration is not being shown here --
       // step1/FlatFileListWriter/Delegate Writer
       container.StepScopeRegistration<IItemWriter<EmployeeDetailBO>,
         FlatFileItemWriter<EmployeeDetailBO>>("step1/FlatFileWriter")
           .Property("Resource")
            .Resource("#{settings['BA_FLAT_FILE_WRITER.step1.FlatFileWriter.FILENAME_OUT']}")
           .Property("Encoding").Value(Encoding.GetEncoding("UTF-8"))
           .Property("LineAggregator")
            .Reference<ILineAggregator<EmployeeDetailBO>>("step1/FlatFileWriter/LineAggregator")
           .Register();

       // Line aggregator
       container.StepScopeRegistration<ILineAggregator<EmployeeDetailBO>,
         FormatterLineAggregator<EmployeeDetailBO>>("step1/FlatFileWriter/LineAggregator")
           .Property("Format")
            .Value("{0},{1},{2},{3:yyyy-MM-dd},{4},{5},{6},{7},{8},{9}")
           .Property("FieldExtractor")
            .Reference<IFieldExtractor<EmployeeDetailBO>>("step1/FlatFileWriter/FieldsExtractor")
           .Register();

       // Fields Extractor
       container.StepScopeRegistration<IFieldExtractor<EmployeeDetailBO>,
         PropertyFieldExtractor<EmployeeDetailBO>>("step1/FlatFileWriter/FieldsExtractor")
    .Property("Names").LateBinding<string[]>("EmpId,EmpName,Name,EmpDob,EmpSalary," +
                    "EmailId,BuildingNo,StreetName,City,State")
           .Register();
}
```



Note

- `FormatterLineAggregator` requires a `Summer.Batch.Infrastructure.Item.File.Transform.IFieldExtractor<in T>` implementation at initialization time;
  The `IFieldExtractor` is in charge of converting a business object into an array of its parts (array of values, build using the object properties); Summer Batch provides several implementations:

  - `S.B.Infrastructure.Item.File.Transform.PropertyFieldExtractor<T>`: this is the implementation being used in the example; it retrieves values from property names (using `Names` property); Examining the line :

    ```
    .Property("Names").LateBinding<string[]>("EmpId,EmpName,Name,EmpDob,EmpSalary,"
    ```

```
        +"EmailId,BuildingNo,StreetName,City,State")
```

we see that all the `EmployeeDetailBO` properties are being selected to be written to target flat file; The order is significant. These properties values will be passed in that order to `string.Format` method which is used by `FormattedLineAggregator`. The format being used

```
.Property("Format").Value("{0},{1},{2},{3:yyyy-MM-dd},{4},{5},{6},{7},{8},{9}")
```

indicates that all selected properties will be effectively written to targeted flat file.

- `S.B.Infrastructure.Item.File.Transform.PassThroughFieldExtractor<object>`: this implementation returns the business object as an array; see api doc for details;

Reading from and writing to RDBMS

Reading from a database

Support for reading from a database is brought by

`Summer.Batch.Infrastructure.Item.Database.DataReaderItemReader<T>` class.

This is an all purpose database reader, usable with any rdbms for which a `System.Data.Common.DbProviderFactory` can be provided.

`DataReaderItemReader` requires the following properties to be set at initialization time :

- ConnectionString : a `System.Configuration.ConnectionStringSettings` instance; connection string is used to provide all required details needed to connect to a given database. These details are usually being stored in an application XML configuration file;
- Query : a string representing the SQL query to be executed against the database; Obviously, only SELECT SQL statements should be used as query in a database reader. Externalizing these SQL queries in a resource file (.resx) is recommended;
- RowMapper: instance of a `Summer.Batch.Data.RowMapper<out T>`, in charge of converting a row from resultset returned by the query execution to a business object of type T.

In addition, if query contains any parameter, one need to supply a
Parameter Source (class that implements

`Summer.Batch.Data.Parameter.IQueryParameterSource` interface).

Supported syntax for query parameters : the query parameters are identified either with ':' or '@' prefix;

Example 6.9. Query parameters supported syntax examples

(both query are valid and equivalent):

- ```
  select CODE,NAME,DESCRIPTION,DATE from BA_SQL_READER_TABLE_1
       where CODE = :Code
  ```

- ```
  select CODE,NAME,DESCRIPTION,DATE from BA_SQL_READER_TABLE_1
       where CODE = @Code
  ```

Two implementations of `IQueryParameterSource` are provided in Summer Batch :

- `Summer.Batch.Data.Parameter.DictionaryParameterSource`: parameters for queries are stored in a dictionary ; matching with query parameters will be done using dictionary entries keys;
- `Summer.Batch.Data.Parameter.PropertyParameterSource`: `PropertyParameterSource` constructor requires a business object as argument. the query parameters will be filled by the business object properties, matching will be done using the properties names.

The query used in example is

```
select CODE,NAME,DESCRIPTION,DATE from BA_SQL_READER_TABLE_1
```

Records are read from BA_SQL_READER_TABLE_1 table, whose DDL is (target database is MS SQL Server) :

```
CREATE TABLE [dbo].[BA_SQL_READER_TABLE_1]  (
  [IDENTIFIER] BIGINT IDENTITY(1,1) NOT NULL,
  [CODE] INT ,
  [NAME] VARCHAR(30) ,
  [DESCRIPTION] VARCHAR(40) ,
  [DATE] DATE
  )
;
```

Result from this query execution will be stored in the following `DatasourceReaderBO` business object:

Example 6.10. `DataReaderItemReader` target business object

```
using System;

namespace Com.Netfective.Bluage.Business.Batch.Bos
{
    /// <summary>
    /// Entity DatasourceReaderBO.
    /// </summary>
    [Serializable]
    public class DatasourceReaderBO
    {
        /// <summary>
        /// Property Code.
        /// </summary>
        public int? Code { get; set; }

        /// <summary>
        /// Property Name.
        /// </summary>
        public string Name { get; set; }

        /// <summary>
        /// Property Description.
        /// </summary>
        public string Description { get; set; }

        /// <summary>
```

```
        /// Property Date.
        /// </summary>
        public DateTime? Date { get; set; }

    }
}
```

Mapping between resultset rows and given business object is done by the following RowMapper:

Example 6.11. DataReaderItemReader target business object RowMapper

```
using Summer.Batch.Data;
using System;

namespace Com.Netfective.Bluage.Business.Batch.Bos.Mappers
{
    /// <summary>
    /// Utility class defining a row mapper for SQL readers.
    /// </summary>
    public static class DatasourceReaderSQLReaderMapper
    {
        /// <summary>
        /// Row mapper for <see cref="DatasourceReaderBO" />.
        /// </summary>
        public static readonly RowMapper<DatasourceReaderBO> RowMapper =
        (dataRecord, rowNumber) =>
        {
            var wrapper = new DataRecordWrapper(dataRecord);
            return new DatasourceReaderBO
            {
                Code = wrapper.Get<int?>(0),
                Name = wrapper.Get<string>(1),
                Description = wrapper.Get<string>(2),
                Date = wrapper.Get<DateTime?>(3),
            };
        };
    }
}
```

Now let's review how to configure a database reader. First, the job XML file:

Example 6.12. DataReaderItemReader declaration in the job XML file

```
<step id="DatasourceReader">
 <chunk item-count="1000">
  <reader ref="DatasourceReader/DatasourceReaderRecord" />
  ...
 </chunk>
</step>
```

Unity configuration:

Example 6.13. Formatted flat file writer - sample unity configuration

```
/// <summary>
/// Registers the artifacts required to execute the steps (tasklets, readers, writers, etc.)
/// </summary>
/// <param name="container">the unity container to use for registrations</param>
public override void LoadArtifacts(IUnityContainer container)
{
    ConnectionStringRegistration.Register(container);
    RegisterDatasourceReader(container);
}
```

```
/// <summary>
/// Registers the artifacts required for step DatasourceReader.
/// </summary>
/// <param name="container">the unity container to use for registrations</param>
private void RegisterDatasourceReader(IUnityContainer container)
{
 //Connection string
 var readerConnectionstring =
  System.Configuration.ConfigurationManager.ConnectionStrings["ReaderConnection"];

    // Reader - DatasourceReader/DatasourceReaderRecord
    container.StepScopeRegistration<IItemReader<DatasourceReaderBO>,
     DataReaderItemReader<DatasourceReaderBO>>("DatasourceReader/DatasourceReaderRecord")
        .Property("ConnectionString").Instance(readerConnectionstring)
        .Property("Query").Value(SqlQueries.DatasourceReader_SQL_QUERY)
        .Property("RowMapper").Instance(DatasourceReaderSQLReaderMapper.RowMapper)
        .Register();

  // ... Processor and writer registration not being shown here
```

Note

- In this example, the connection string is read from the default application configuration file, using System.Configuration.ConfigurationManager. Here is the corresponding XML configuration (points at a MS SQL Server database):

```
<?xml version="1.0" encoding="utf-8" ?>
<connectionStrings>
  <add name="ReaderConnection"
   providerName="System.Data.SqlClient"
   connectionString="Data Source=(LocalDB)\v11.0;
     AttachDbFilename=|DataDirectory|\data\BA_SQL_Reader.mdf;Integrated Security=True" />
</connectionStrings>
```

- The query is read into a resource file (SqlQueries.resx); SqlQueries class is the corresponding designer artifact created by Visual Studio to wrap the underlying resource.

Writing to a database

`Summer.Batch.Infrastructure.Item.Database.DatabaseBatchItemWriter<T>` is able to write a collection of business objects to the target database, using a INSERT or UPDATE SQL statement.
The following properties must be set at initialization time:

- ConnectionString : a System.Configuration.ConnectionStringSettings instance; connection string is used to provide required details needed to connect to a given database. These details are usually being stored in an application XML configuration file;

- Query : a string representing SQL query to be executed on the database; To write to a database, only INSERT or UPDATE SQL statements should be used. Externalizing SQL queries in a resource file (.resx) is recommended; As for reader, query parameters will be prefixed either by ':' or '@'.

- DbParameterSourceProvider : a class that implements `Summer.Batch.Data.Parameter.IQueryParameterSourceProvider<in T>` interface, in charge of filling query parameters with appropriate values, usually from a business object.

  The method to be implemented is

  `IQueryParameterSource CreateParameterSource(T item)` : query parameters will be fed by consuming `Summer.Batch.Data.Parameter.IQueryParameterSource`.

Summer Batch provides an implementation for `IQueryParameterSourceProvider` interface :

`Summer.Batch.Data.Parameter.PropertyParameterSourceProvider<in T>`

Given an business object of type T, this class will provide a `Summer.Batch.Data.Parameter.PropertyParameterSource` that maps query parameters to the business object properties, on a naming convention basis.

An extra property can be used to refine the writer behavior :

- AssertUpdates : a flag to indicate whether to check if database rows have actually been updated; defaults to true (Recommended mode);

Our writer will be using the following query

```
INSERT INTO BA_SQL_WRITER_TABLE (CODE,NAME,DESCRIPTION,DATE)
   VALUES (:code,:name,:description,:date)
```

, that will write records into the BA_SQL_WRITER_TABLE whose DDL -- for MS SQL Server -- is

```
CREATE TABLE [dbo].[BA_SQL_WRITER_TABLE]  (
  [IDENTIFIER] BIGINT IDENTITY(1,1) NOT NULL,
  [CODE] INT ,
  [NAME] VARCHAR(30) ,
  [DESCRIPTION] VARCHAR(40) ,
  [DATE] DATE
  )
;
```

Each row written will correspond to an instance of the following business object

Example 6.14. `DatabaseBatchItemWriter` target business object

```
using System;

namespace Com.Netfective.Bluage.Business.Batch.Datasource.Bos
{
    /// <summary>
    /// Entity DatasourceWriterBO.
    /// </summary>
    [Serializable]
    public class DatasourceWriterBO
    {
        /// <summary>
        /// Property Code.
        /// </summary>
        public int? Code { get; set; }

        /// <summary>
        /// Property Name.
        /// </summary>
        public string Name { get; set; }

        /// <summary>
        /// Property Description.
        /// </summary>
        public string Description { get; set; }
```

```
        /// <summary>
        /// Property Date.
        /// </summary>
        public DateTime? Date { get; set; }

    }
}
```

Eventually, mapping between business object and query parameters is achieved by using a `PropertyParameterSourceProvider`.

Now, we need to configure the writer; Firstly in job XML file :

Example 6.15. `DatabaseBatchItemWriter` declaration in the job XML file

```
<step id="DatasourceWriter">
 <chunk item-count="1000">
  ...
  <writer ref="DatasourceWriter/WriteRecodDatasource" />
 </chunk>
</step>
```

Unity configuration:

Example 6.16. Database batch writer - sample unity configuration

```
/// <summary>
/// Registers the artifacts required for step DatasourceWriter.
/// </summary>
/// <param name="container">the unity container to use for registrations</param>
private void RegisterDatasourceWriter(IUnityContainer container)
{
 //Connection string
 var writerConnectionstring =
  System.Configuration.ConfigurationManager.ConnectionStrings["WriterConnection"];

    //... reader and processor registration not shown here

    // Writer - DatasourceWriter/WriteRecodDatasource
    container.StepScopeRegistration<IItemWriter<DatasourceWriterBO>,
      DatabaseBatchItemWriter<DatasourceWriterBO>>("DatasourceWriter/WriteRecodDatasource")
        .Property("ConnectionString").Instance(writerConnectionstring)
        .Property("Query").Value(SqlQueries.WriteRecodDatasource_S01_DatasourceWriter_SQL_QUERY)
        .Property("DbParameterSourceProvider")
         .Reference<PropertyParameterSourceProvider<DatasourceWriterBO>>()
        .Register();
}
```

Note

As in the database reader example:

- Connection details are read from an application XML config file;
- Query is read from a resource file (.resx);

Database Support

Summer Batch currently supports following three RDBMS:

RDBMS	Provider Names
Microsoft® SQL Server	`System.Data.SqlClient`

RDBMS	Provider Names
Oracle® Database	System.Data.OracleClient, Oracle.ManagedDataAccess.Client, Oracle.DataAccess.Client
IBM® DB2	IBM.Data.DB2

Summer.Batch.Data.IDatabaseExtension interface allows extending Summer Batch to support more RDBMS or provider names. Implementations of this interface present in a referenced assembly will automatically be detected and registered. It has three properties that require a getter:

ProviderNames
: An enumeration of the invariant provider names supported by this extension.

PlaceHolderGetter
: An instance of IPlaceHolderGetter is used to replace parameters in queries with placeholders. All SQL queries should use either "@" or ":" to prefix parameters, the placeholder getter will be used to transform the query so that it uses placeholders required by the provider.

Incrementer
: An instance of IDataFieldMaxValueIncrementer is used to retrieve unique ids for different batch entities, such as job or step executions. Best way to generate unique ids is dependent on the specified RDBMS, thus the extension is required to provide an incrementer.

The following example show how to add support for PostgreSQL using Npgsql provider:

Example 6.17. Adding support for other RDBMS : the PostgreSQL example

```
public class PostgreSqlExtension : IDatabaseExtension
{
    public IEnumerable<string> ProviderNames
    {
        get { return new[] { "Npgsql" }; }
    }

    public IPlaceholderGetter PlaceholderGetter
    {
        get { return new PlaceholderGetter(name => ":" + name, true); }
    }

    public IDataFieldMaxValueIncrementer Incrementer
    {
        get { return new PostgreSqlIncrementer(); }
    }
}

public class PostgreSqlIncrementer : AbstractSequenceMaxValueIncrementer
{
    protected override string GetSequenceQuery()
    {
        return string.Format("select nextval('{0}')", IncrementerName);
    }
}
```

In addition to providing a complete support for an additional RDBMS, job repository DDL's should be adapted to the targeted RDBMS.

Some required DDL adjustments are:

- adapting types names to RDBMS types (BIGINT vs NUMBER, etc.)

- adapting sequence support mechanisms (which are very RDBMS dependant)

Caution

We do NOT officially support PostgreSQL in Summer Batch, but for the additional RDBMS support, examples are provided in corresponding appendix section, DDL scripts for PostgreSQL.

Chapter 7. Using Advanced Features

Reading and writing EBCDIC files using COBOL copybooks

COBOL batches frequently involves dealing with EBCDIC files using copybooks. Modernizing these batches requires ability to read and write to these files. Two classes have been developed to support EBCDIC Read/Write:

- `Summer.Batch.Extra.Ebcdic.EbcdicFileReader`: to read from an EBCDIC file;
- `Summer.Batch.Extra.Ebcdic.EbcdicFileWriter`: to write to an EBCDIC file;

Both classes require a XML version of a COBOL copybook provided at run time.

Using the `EbcdicFileReader`

To read records from an EBCDIC file, using an EbcdicFileReader, the following elements must be provided :

- a XML version of the needed COBOL copybook (Copybook property);
- a class in charge of transforming EBCDIC record into a business object (EbcdicReaderMapper property): this class must implement `Summer.Batch.Extra.Copybook.IEbcdicReaderMapper<T>` interface ;
- a path to EBCDIC file (Resource property).

Caution

The two elements (copybook XML file and EBCDIC reader mapper class) are mandatory.

The COBOL copybook is used by reader to extract records from the EBCDIC file.

Example 7.1. Sample XML copybook export

```
<?xml version="1.0" encoding="ASCII"?>
<FileFormat ConversionTable="IBM037" dataFileImplementation="IBM i or z System"
 distinguishFieldSize="0" newLineSize="0" headerSize="0">
  <RecordFormat cobolRecordName="BA_EBCDIC_READER" distinguishFieldValue="0">
    <FieldFormat Decimal="0" DependingOn="" ImpliedDecimal="true" Name="CODE" Occurs="1"
     Picture="S9(5)" Signed="true" Size="5" Type="3" Value=""/>
    <FieldFormat Decimal="0" DependingOn="" ImpliedDecimal="true" Name="NAME" Occurs="1"
     Picture="X(30)" Signed="false" Size="30" Type="X" Value=""/>
    <FieldFormat Decimal="0" DependingOn="" ImpliedDecimal="true" Name="DESCRIPTION" Occurs="1"
     Picture="X(40)" Signed="false" Size="40" Type="X" Value=""/>
    <FieldFormat Decimal="0" DependingOn="" ImpliedDecimal="true" Name="DATE" Occurs="1"
     Picture="S9(8)" Signed="true" Size="4" Type="B" Value=""/>
  </RecordFormat>
</FileFormat>
```

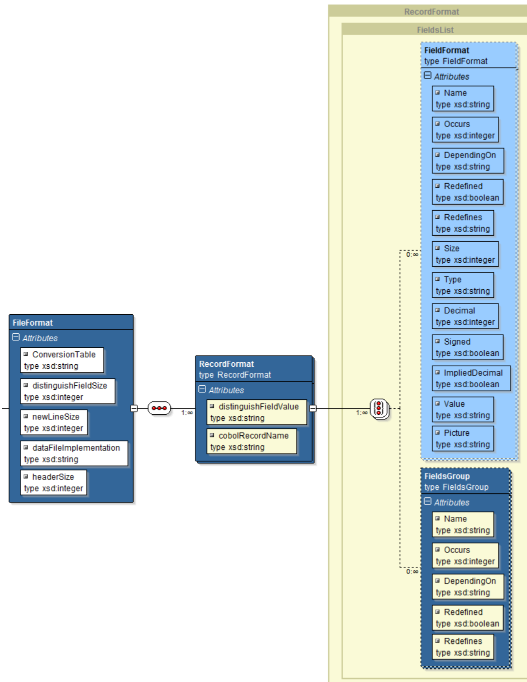
Note

ConversionTable (=encoding) is specified in the copybook XML file: this attribute must have a property that can be understood by the C# API. This requires use the C# encoding names (which differ from java encoding names convention, for

example, java uses "Cp037" where C# uses "IBM037"). `System.Text.Encoding` has `GetEncodings` method to list all available encodings.

The XML copybook file must be compliant with XSD (full XSD source is given in dedicated appendix section)

Figure 7.1. EBCDIC File Format XML schema



The corresponding bound C# classes are located in `Summer.Batch.Extra.Copybook` namespace:

- `CopybookElement.cs`
- `FieldFormat.cs`
- `FieldsGroup.cs`
- `FileFormat.cs`
- `IFieldsList.cs`
- `RecordFormat.cs`

Afterwards the EBCDIC reader mapper is used to transform the records into a business objects. Below is a business object whose properties will be mapped to the EBCDIC record described by the XML copybook above.

Example 7.2. Sample business object to which EBCDIC records will be mapped

```
using System;

namespace Com.Netfective.Bluage.Business.Batch.Ebcdic.Bos
{
    /// <summary>
```

```
    /// Entity EbcdicFileBO.
    /// </summary>
    [Serializable]
    public class EbcdicFileBO
    {
        /// <summary>
        /// Property Code.
        /// </summary>
        public int? Code { get; set; }

        /// <summary>
        /// Property Name.
        /// </summary>
        public string Name { get; set; }

        /// <summary>
        /// Property Description.
        /// </summary>
        public string Description { get; set; }

        /// <summary>
        /// Property Date.
        /// </summary>
        public DateTime? Date { get; set; }

    }
}
```

Now, Define a mapper in charge of transforming records into business objects. This mapper must implement `Summer.Batch.Extra.Ebcdic.IEbcdicReaderMapper<T>` interface. `Summer.Batch.Extra.Ebcdic.AbstractEbcdicReaderMapper<T>` is super class to inherit from in order to craft EBCDIC reader mapper quickly (see example below).

Example 7.3. Sample EBCDIC reader mapper

```
using Com.Netfective.Bluage.Business.Batch.Ebcdic.Bos;
using Summer.Batch.Extra.Ebcdic;
using System.Collections.Generic;

namespace Com.Netfective.Bluage.Business.Batch.Ebcdic.Bos.Mappers
{
 ///<summary>
 /// Ebcdic mapper for the EbcdicFileBO class.
 ///</summary>
    public class EbcdicFileEbcdicMapper : AbstractEbcdicReaderMapper<EbcdicFileBO>
    {
        private const int Code = 0;
        private const int Name = 1;
        private const int Description = 2;
        private const int Date = 3;

  ///<summary>
  /// Map a collection of properties to a EbcdicFileBO record.
  ///</summary>
  /// <param name="values"> list of values to be mapped</param>
  /// <param name="itemCount"> item count; will be used as identifier
  /// if this makes sense for the target class.</param>
  /// <returns>the EbcdicFileBO record build upon the given list of values.</returns>
        public override EbcdicFileBO Map(IList<object> values, int itemCount)
        {
            var record = new EbcdicFileBO
            {
                Code = (int) ((decimal) values[Code]),
                Name = ((string) values[Name]),
```

```
                Description = ((string) values[Description]),
                Date = ParseDate(values[Date]),
            };
            return record;
        }
    }
}
```

The EbcdicFileReader is then declared as any other reader in the job XML file :

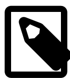
Example 7.4. `EbcdicFileReader` declaration in the job XML file

```
<step id="EbcdicReader">
    <chunk item-count="1000">
        <reader ref="EbcdicReader/EbcdicFileReader" />
        …
    </chunk>
</step>
```

And here is the corresponding Unity configuration part :

Example 7.5. `EbcdicFileReader` Unity configuration

```
…
/// <summary>
/// Registers the artifacts required for step EbcdicReader.
/// </summary>
/// <param name="container">the unity container to use for registrations</param>
private void RegisterEbcdicReader(IUnityContainer container)
{
    // Reader - EbcdicReader/EbcdicFileReader
    container.StepScopeRegistration<IItemReader<EbcdicFileBO>, EbcdicFileReader<EbcdicFileBO>>
        ("EbcdicReader/EbcdicFileReader")
        .Property("Resource").Resource("#{settings['BA_EBCDIC_READER.EbcdicReader.FILENAME_IN']}")
        .Property("Copybook").Resource("#{settings['BA_EBCDIC_READER.EbcdicReader.COPYBOOK_IN']}")
        .Property("EbcdicReaderMapper")
            .Reference<IEbcdicReaderMapper<EbcdicFileBO>>("EbcdicReader/EbcdicFileReader/Mapper")
        .Register();
…
```



Note

Please note:

- EBCDIC file reader is registered within a step scope, using the `StepScopeRegistration` extension;
- EBCDIC file reader registration is done using the same name as it was declared in the job XML file (that is "EbcdicReader/EbcdicFileReader");
- The

  ```
  Property("Resource")
      .Resource("#{settings['BA_EBCDIC_READER.EbcdicReader.FILENAME_IN']}")
  ```

  call indicates that the property named "Resource" will be filled with value read in Settings.config XML file. Here is the corresponding Settings.config file content :

  ```
  <?xml version="1.0" encoding="utf-8" ?>
  <appSettings>
    <add key="BA_EBCDIC_READER.EbcdicReader.FILENAME_IN"
     value="data\inputs\BA_EBCDIC_READER.data" />
  ```

```
  <add key="BA_EBCDIC_READER.EbcdicReader.COPYBOOK_IN"
   value="data\copybooks\BA_EBCDIC_READER.fileformat" />
</appSettings>
```

Using the `EbcdicFileWriter`

To write records to an EBCDIC file using an EbcdicFileWriter, the following elements should be provided:

- list of COBOL copybooks XML versions (Copybooks property);
- class in charge of transforming EBCDIC record into business object (EbcdicWriterMapper property): one can use the `Summer.Batch.Extra.Ebcdic.EbcdicWriterMapper` class or a custom sub-class inherited from it;
- path to the targeted EBCDIC file (Resource property).

Caution

Writer takes a list of XML copybooks but only uses one copybook at a time; writer has a method (`ChangeCopyBook`) to change the current copybook being used.

These three elements (copybook XML files list, EBCDIC writer mapper class and path to targeted resource) are mandatory.

Let's review the corresponding configurations. The writer must be declared in the job XML file (as any other writer):

Example 7.6. `EbcdicFileWriter` declaration in the job XML file

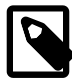
```
<step id="EBCDIC_WRITER">
    <chunk item-count="1000">
        …
        <writer ref="EBCDIC_WRITER/EbcdicFileWriter" />
    </chunk>
</step>
```

Unity configuration:

Example 7.7. `EbcdicFileWriter` Unity configuration

```
…
/// <summary>
/// Registers the artifacts required for step EBCDIC_WRITER.
/// </summary>
/// <param name="container">the unity container to use for registrations</param>
private void RegisterEBCDIC_WRITER(IUnityContainer container)
{
  …

    // Writer - EBCDIC_WRITER/EbcdicFileWriter
    container
       .StepScopeRegistration<IItemWriter<EbcdicFileBO>, EbcdicFileWriter<EbcdicFileBO>>("EBCDIC_WRITER/EbcdicFileWriter")
       .Property("Resource").Resource("#{settings['BA_EBCDIC_WRITER.EBCDIC_WRITER.EbcdicFileWriter.FILENAME_OUT']}")
       .Property("Copybooks").Resources("#{settings['BA_EBCDIC_WRITER.EBCDIC_WRITER.EbcdicFileWriter.COPYBOOK_OUT']}")
       .Property("EbcdicWriterMapper").Reference<EbcdicWriterMapper>("EBCDIC_WRITER/EbcdicFileWriter/Mapper")
       .Register();

    // EBCDIC writer mapper for EBCDIC_WRITER/EbcdicFileWriter
    container
       .StepScopeRegistration<EbcdicWriterMapper>("EBCDIC_WRITER/EbcdicFileWriter/Mapper")
       .Register();
```

```
…
```



Note

- EBCDIC file writer is registered within a step scope, using the `StepScopeRegistration` extension;
- EBCDIC file writer registration is done using the same name as it was declared in the job XML file (that is "EBCDIC_WRITER/EbcdicFileWriter");
- Regarding the list of copybooks resources loading :

  ```
  .Property("Copybooks").Resources("#{settings[…]}")
  ```

  `.Resources()` call (note the extra "s") is able to load a list of resource paths into a semicolon-separated path string.
- `Summer.Batch.Extra.Ebcdic.EbcdicWriterMapper` automatically convert a business object into a list of values that can be written in an EBCDIC record (Refer to `Summer.Batch.Extra.Ebcdic.EbcdicWriterMapper#Map` method). The automatic mapping between business object properties and copybook records is made on a name convention basis. The property name must be equal to FieldFormat Name attribute, *converted to camel case*.

  e.g.

  ```
  <FieldFormat Decimal="0" DependingOn="" ImpliedDecimal="true"
      Name="DESCRIPTION" Occurs="1" Picture="X(40)" Signed="false"
      Size="40" Type="X" Value=""/>
  ```

  will automatically map to the property named "Description" (= camel case version of "DESCRIPTION")

  ```
  /// <summary>
  /// Property Description.
  /// </summary>
  public string Description { get; set; }
  ```

  To make sure you are using correct names, in order to guarantee the automatic mapping between records and business object properties, you can use the ToCamelCase method from the `Summer.Batch.Extra.Ebcdic.AbstractEbcdicMapper` class;

FTP operations support

Two dedicated FTP operations tasklets are provided in Summer Batch:

- `Summer.Batch.Extra.FtpSupport.FtpPutTasklet`: provides FTP put operation support;
- `Summer.Batch.Extra.FtpSupport.FtpGetTasklet`: provides FTP get operation support.

Both are using the `System.Net.FtpWebRequest` FTP client.

FTP put operations

Using `FtpPutTasklet` you can put a given file into a FTP remote directory. The following properties are mandatory (need to be set at initialization time):

- FileName : path to the file that will be put on the FTP remote directory;

- Host : FTP host (name or I.P. address);
- Username : the user name to connect to FTP host;
- Password : password for the above FTP user.

The following properties are optional (Default value is provided, however can set at initialization time):

- Port : FTP host port. Defaults to 21;
- RemoteDirectory : path to FTP remote directory. Defaults to empty string, meaning that the default remote directory is the FTP root directory.

Configuring the `FtpPutTasklet` in the job XML file:

Example 7.8. `FtpPutTasklet` usage in the job XML file

```
<step id="FtpPutStep">
 <batchlet ref="FtpPutBatchlet" />
</step>
```

and sample Unity configuration is as below:

Example 7.9. `FtpPutTasklet` Unity configuration

```
…
/// <summary>
/// Registers the artifacts required for step FtpPutStep.
/// </summary>
/// <param name="container">the unity container to use for registrations</param>
private void RegisterFtpPutStep(IUnityContainer container)
{
            container.StepScopeRegistration<ITasklet, FtpPutTasklet>("FtpPutBatchlet")
                .Property("Host").Value("myftp.mycompany.com")
                .Property("FileName").Resource("#{settings['FtpPutBatchlet.FILENAME_IN']}")
                .Property("Username").Value("anonymous")
                .Property("Password").Value("123soleil")
                .Register();
…
```

FTP get operations

Using the `FtpGetTasklet`, you can get a given set of files from a FTP remote directory. The following properties are mandatory (need to be set at initialization time):

- Host : FTP host (name or I.P. address);
- Username : the user name to connect to the FTP host;
- Password : password for FTP user;
- FileNamePattern : a filename pattern, similar in form to what DirectoryInfo.GetFiles(string) supports to filter the files to be downloaded from the FTP remote directory;
- LocalDirectory : path to local directory where downloaded files will be stored;
- AutoCreateLocalDirectory : boolean flag to control whether the local directory should be automatically created if non-existent (defaults to true); If AutoCreateLocalDirectory is set

to false and LocalDirectory does not exist, a `System.IO.DirectoryNotFoundException` will be thrown at run time.

- RemoteDirectory : path to FTP remote directory to download files from;

The following properties are optional (Default value is provided, however can set at initialization time):

- Port : FTP host port; Defaults to 21;
- AutoCreateLocalDirectory : defaults to true; see LocalDirectory item above for the meaning of this flag;
- DownloadFileAttempts : number of file download attempts before giving up; Defaults to 12;
- RetryIntervalMilliseconds : Time in milliseconds to wait for a retry (after a failure); defaults to 300000;
- DeleteLocalFiles : boolean flag (defaults to true) to indicate whether local existing files (from a prior download for example) will be deleted before attempting a fresh download. FileNamePattern property will be used to filter files to be deleted.

Configuring the `FtpGetTasklet` in the job XML file:

Example 7.10. `FtpGetTasklet` usage in the job XML file

```
<step id="FtpGetStep">
 <batchlet ref="FtpGetBatchlet" />
</step>
```

and here is a sample Unity configuration :

Example 7.11. `FtpGetTasklet` Unity configuration

```
…
/// <summary>
/// Registers the artifacts required for step FtpGetStep.
/// </summary>
/// <param name="container">the unity container to use for registrations</param>
private void RegisterFtpGetStep(IUnityContainer container)
{
            container.StepScopeRegistration<ITasklet, FtpGetTasklet>("FtpGetBatchlet")
                .Property("Host").Value("myftp.mycompany.com")
                .Property("FileNamePattern").Value("*.txt")
                .Property("Username").Value("anonymous")
                .Property("Password").Value("123soleil")
                .Property("LocalDirectory").Value("C:/temp/downloads")
                .Property("RemoteDirectory").Value("/my/remote/directory")
                .Register();
…
```

Email sending support

Using `EmailTasklet` (full name : `Summer.Batch.Extra.EmailSupport.EmailTasklet`) you can send email by SMTP protocol. This tasklet relies on `System.Net.Mail.SmtpClient` to perform this operations.

The following properties are mandatory (need to be set at initialization time):

- Host : name or I.P. address of the SMTP server used for sending mails;
- From : from email address for sending mails;
- Subject : subject of mail to be sent;
- Body : resource path to mail body;
- At least one recipient must be specified: recipients are stored in three separate lists:
  - To : list of direct recipients addresses for mail;
  - Cc : list of copy recipients addresses for mail;
  - Bcc : list of hidden copy recipients addresses for mail;

  At least one of these lists must be non-empty.

Optional properties (Default value is provided, however can set at initialization time):

- Username : to be provided if the SMTP server requires a user name; No default value;
- Password : to be provided if the SMTP server requires a password for the username; No default value;
- InputEncoding : input encoding of the mail body (read); defaults to the platform default encoding;
- Encoding : encoding of the mail to be sent (write); defaults to the platform default encoding;
- Port : SMTP server port; defaults to 25;
- LineLength : zero or a positive integer to indicate the maximum line length used for reading the mail body; if 0, the whole body will be read in a single pass (default option); otherwise, the body will be read line by line of the specified LineLength size.

Configuring the `EmailTasklet` in the job XML file:

Example 7.12. `EmailTasklet` usage in the job XML file

```
<step id="EmailStep">
 <batchlet ref="EmailBatchlet" />
</step>
```

and below is a sample Unity configuration :

Example 7.13. `EmailTasklet` Unity configuration

```
…
/// <summary>
/// Registers the artifacts required for step EmailStep.
/// </summary>
/// <param name="container">the unity container to use for registrations</param>
private void RegisterEmailStep(IUnityContainer container)
{
            container.StepScopeRegistration<ITasklet, EmailTasklet>("EmailBatchlet")
                .Property("Host").Value("mysmptserver")
                .Property("Body").Resource("#{settings['EmailBatchlet.BODY']}")
                .Property("From").Value("anonymous@mycompany.com")
                .Property("Subject").Value("test email from batch system")
                .Property("To").LateBinding<string[]>("t.masson@bluage.com")
                .Property("Encoding").Value(Encoding.GetEncoding("UTF-8"))
```

```
                .Property("InputEncoding").Value(Encoding.GetEncoding("IBM01047"))
                .Property("LineLength").Value(80)
                .Register();
…
```

Empty file check support

`Summer.Batch.Extra.EmptyCheckSupport.EmptyFileCheckTasklet` is dedicated to check if a given file is empty or not. It returns an ExitStatus "EMPTY" if the file is empty or absent, "NOT_EMPTY" if it exists and is not empty.

Only mandatory property is FileToCheck; it points at the target file resource to be checked for existence/emptiness.

Configuring the EmptyFileCheckTasklet in the job XML file:

Example 7.14. Typical `EmptyFileCheckTasklet` usage in the job XML file

```
<step id="S01_EmptyFileCheck">
 <batchlet ref="S01_EmptyFileCheckBatchlet" />
 <next on="NOT_EMPTY" to="S02_Writer" />
 <next on="EMPTY" to="S03_Writer" />
</step>
<step id="S02_Writer" next="S03_Writer">
 <chunk item-count="1000">
  <reader ref="S02_Writer/Reader" />
  <writer ref="S02_Writer/Writer" />
 </chunk>
</step>
<step id="S03_Writer">
 <chunk item-count="1000">
  <reader ref="S03_Writer/Reader" />
  <writer ref="S03_Writer/Writer" />
 </chunk>
</step>
```

Note

Recommendation: Usage should use of the returned ExitStatus as a switch to organize the job execution flow, as demonstrated by the example above.

Example 7.15. Sample Unity configuration for a `EmptyFileCheckTasklet`

```
…
// Step S01_EmptyFileCheck - Empty file check step
private void RegisterS01_EmptyFileCheckTasklet(IUnityContainer container)
{
    container.StepScopeRegistration<ITasklet,EmptyFileCheckTasklet>("S01_EmptyFileCheckBatchlet")
    .Property("FileToCheck").Resource("#{settings['BA_EMPTY_FILE_SB.S01_EmptyFileCheck.FILENAME_IN']}")
    .Register();
}
…
```

Sql Script Runner support

`Summer.Batch.Extra.SqlScriptSupport.SqlScriptRunnerTasklet` is dedicated to launch an external SQL script.

Two below properties are mandatory: as usual for SQL processes, a ConnectionStringSettings must be supplied, and the resource to be read (i.e. the script file) must also be supplied.

- ConnectionStringSettings
- the resource to be read (i.e. the script file) must also be supplied.

Configuring the SqlScriptRunnerTasklet in the job XML file:

Example 7.16. Typical `SqlScriptRunnerTasklet` usage in the job XML file

```
<step id="SQLScriptWriter">
 <batchlet ref="SQLScriptWriterBatchlet" />
</step>
```

Example 7.17. Sample Unity configuration for a `SqlScriptRunnerTasklet`

```
…
// Step SQLScriptWriter - SQL script step
private void RegisterSQLScriptWriterTasklet(IUnityContainer container)
{
    container.StepScopeRegistration<ITasklet, SqlScriptRunnerTasklet>("SQLScriptWriterBatchlet")
        .Property("ConnectionStringSettings").Reference<ConnectionStringSettings>("Default")
        .Property("Resource").Resource("#{settings['BA_SQLScriptWriter.SQLScriptWriter.FILENAME']}")
        .Register();
}
…
```

Sort Tasklet

`Summer.Batch.Extra.Sort.SortTasklet` tasklet can sort or merge files using the syntax of DFSORT. It is useful when modernizing batches from z/OS mainframes. Only supported features are described in this section, see the DFSORT documentation for more details on the legacy syntax.

Specifying input and output files. The input files are specified using `Input` property, which expects a list of resources, and output file is specified using `Output` property, which expect a resource. The encoding of input files can be set using `Encoding` property. If no encoding is specified, default encoding is used. All input files must have same encoding, which will be preserved in the output. If input files have header, it must be specified using `HeaderSize` property; this header will be reproduced in output file.

Reading and writing records. A record is a unit of information that can be sorted. A sorted tasklet supports three types of records: (1) records separated by a character string (e.g., records separated by new line characters), (2) fixed-length records, and (3) records with a record descriptor word (RDW) specifying size. For separated records, the separator must be set with `Separator` property. For fixed-length records, the length must be set with `RecordLength` property. If none of these properties have been set, the tasklet will assume that records have an RDW.

Sorting files. To sort files, *sort card* must be specified with `SortCard` property. If it is set, a comparer will be generated to sort records in input files, otherwise records will be kept in

original order as in the input files. The sort card should respect DFSORT syntax (see section Supported DFSORT Features for more information).

Filtering records. Records can be filtered using `Include` and `Omit`. Each property uses DFSORT syntax for records selection. Only records that are selected by `Include` and not selected by `Omit` will appear in output.

Transforming records. Records can be transformed using `Inrec` and `Outrec` properties. Each property uses DFSORT syntax for record transformation. Records are transformed before and after they are sorted or filtered using `Inrec` and `Outrec` respectively.

Managing duplicates. By default, all identical records will appear in output. The order among these identical elements in unspecified. If `SkipDuplicates` property is set to `true`, only one of identical records will be kept (the selection is also unspecified). It is also possible to sum the lines by setting `Sum` property with a DFSORT sum card.

Supported DFSORT Features

Sort Card

The sort card can either be a list of fields or specify a default format using the following syntax:

```
[FORMAT=<format>,]FIELDS=<fields>[,FORMAT=<format>]
```

The fields should conform to DFSORT syntax, but only the following formats are supported:

- String (CH)
- Zoned (ZD)
- Packed (PD)
- Signed binary (FI) and unsigned binary (BI)

Include and Omit Cards

As with sort cards, include and omit cards can either be a list of conditions or specify a default format:

```
[FORMAT=<format>,]COND=<conditions>[,FORMAT=<format>]
```

The conditions should conform to DFSORT syntax and supports the following features:

Comparable elements

- Fields
- Decimal constants
- Character string constants

Field formats

- String (CH) and sub-string (SS)
- Zoned (ZD)
- Packed (PD)
- Signed binary (FI) and unsigned binary (BI)

Comparison operators

- Equals (EQ)

- Different (NE)
- Greater (GT)
- Greater or equals (GE)
- Lower (LT)
- Lower or equals (LE)

Inrec and Outrec Cards

The record formating in inrec and outrec cards only supports DFSORT "BUILD" syntax with following elements:

- Fields copy (with the same supported formats as for sort cards).
- Character string constants ("C'…'").
- Hexadecimal constants ("X'…'").
- Space constants ("X").
- Numeric editing patterns, including predefined masks ("M0" to "M26"). Signs ("SIGN=") and length ("LENGTH=") statements are supported.

Generation Data Groups (GDG)

Summer Batch allows emulation of main frames' generation data groups (GDG) using a specific resource loader, Summer.Batch.Extra.IO.GdgResourceLoader . Resource loader is in charge of creating instances of IResource when required and is used by ResourceInjectionValue when injecting resources (see section New Injection Parameter Values). To use GDG, Unity loader must register GDG resource loader in the container:

Example 7.18. Registration of the GDG Resource Loader

```
protected override void LoadConfiguration(IUnityContainer container)
{
    base.LoadConfiguration(container);
    container.SingletonRegistration<ResourceLoader, GdgResourceLoader>().Register();
}
```

A generation data group is a group of output files. Files in this group are identified by a reference number: files generated by the current execution of the batch is 1, the file generated by the previous execution is 0, and so on. A file in a GDG is referenced using "gdg://" protocol as follows:

```
gdg://<path>(<number>)[.<extension>]
```

"<path>" is path to the file without extension or generation number, "<number>" is reference number of the file and "<extension>" is extension of the file (which is optional). The final file name will be computed by concatenating the path with generation number and extension if there is one. For instance, if generation number of the GDG for the previous execution was 23, the reference "gdg://data/output(1).txt" would result in the file name "data/outputG0024V00.txt".

Groups can be configured using a specific application setting (in "App.setting") named "gdg-options". It should contain a character string with options for groups used in the batch. Each

group is identified by its path concatenated with "(*)" and its extension (e.g., "gdg://data/output(*).txt"). The grammar for the options is the following:

```
<gdg-options> ::= <gdg-option> | <gdg-option> "," <gdg-options>
<gdg-option>  ::= <group> | <group> "," <options>
<group>       ::= <path> "(*)" | <path> "(*)." <extension>
<options>     ::= <option> | <option> "," <options>
<option>      ::= <limit> | <mode>
<limit>       ::= "limit=" <integer>
<mode>        ::= "mode=empty" | "mode=notempty"
```

"limit" option sets number of file that should be kept in a group and "mode" options specify how files are managed once this limit is reached: in "empty" all files existing before the current execution are deleted while in "notempty" the oldest files are deleted so that the final number of files is the limit. The default mode is "notempty".

Example 7.19. GDG Configuration

Assuming the working directory contains following files:

```
data/customer/reportG0003V00.txt
data/customer/reportG0004V00.txt
data/customer/reportG0005V00.txt
data/commands/summaryG0018V00.txt
data/commands/summaryG0019V00.txt
data/commands/summaryG0020V00.txt
```

that the "gdg-options" setting contains "data/customer/report(*).txt,limit=3,data/commands/summary(*).txt,limit=3,mode=empty" and that the current batch execution will write to "gdg://data/customer/report(1).txt" and "gdg://data/commands/summary(1).txt", at the end of the batch execution the working directory will contain the following files:

```
data/customer/reportG0004V00.txt
data/customer/reportG0005V00.txt
data/customer/reportG0006V00.txt
data/commands/summaryG0021V00.txt
```

Referencing all files in a group. It is possible to reference all existing files in a generation data group by using the "*" wildcard instead of a reference number (e.g., "gdg://data/customer/report(*).txt").

Context Managers

When launching jobs and steps, instances of JobExecution and StepExecution are used by the Summer Batch engine and persisted in the repository. These executions each have a dedicated context that can be used to store data and process variables. In order to give access to theses contexts in a convenient way, several classes were added to the Summer Batch: ContextManager, ContextManagerUnityLoader, AbstractExecutionListener and AbstractService.

ContextManager class

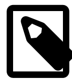
Contexts are basically dictionaries. ContextManager simply enables to store and retrieve from dictionary and gives basic utility methods (particularly related to counter management). There is a single ContextManager class for both job context and step context but two instances will be available, one for the job and other for the step.

It implements `Summer.Batch.Extra.IContextManager` interface, whose code is given below

Example 7.20. `IContextManager` interface contract

```
using Summer.Batch.Infrastructure.Item;

namespace Summer.Batch.Extra
{
    /// <summary>
    /// Interface for context manager
    /// </summary>
    public interface IContextManager
    {
        /// <summary>
        /// Accessors for the context
        /// </summary>
        ExecutionContext Context { get; set; }

        /// <summary>
        /// Stores an object inside the cache
        /// </summary>
        /// <param name="key">object key to store</param>
        /// <param name="record">object to store</param>
        void PutInContext(object key, object record);

        /// <summary>
        /// Check if the key is in the cache
        /// </summary>
        /// <param name="key">key of the object to retrieve</param>
        /// <returns>whether it is in the cache</returns>
        bool ContainsKey(object key);

        /// <summary>
        /// Retrieves an object from the cache
        /// </summary>
        /// <param name="key">key of the object to retrieve</param>
        /// <returns>retrieved object</returns>
        object GetFromContext(object key);

        /// <summary>
        /// Clears the cache
        /// </summary>
        void Empty();

        /// <summary>
        /// Dumps the cache content
        /// </summary>
        /// <returns>the content of the cache as a string</returns>
        string Dump();

        /// <summary>
        /// Sets the value of a named counter
        /// </summary>
        /// <param name="counter">the name of the counter</param>
        /// <param name="value">the new value of the the named counter</param>
        void SetCounter(string counter, long value);

        /// <summary>
        /// Returns the value of a named counter
        /// </summary>
        /// <param name="counter">the name of the counter</param>
        ///<returns>the value of the the named counter</returns>
        long GetCounter(string counter);

        /// <summary>
        ///  Increments the value of a counter by one. If this counter does not yet
        /// exist, it is first created with a value of 0 (thus, the new value is 1).
        /// </summary>
        /// <param name="counter">the name of the counter</param>
        void IncrementCounter(string counter);

        /// <summary>
        ///  Decrements the value of a counter by one. If this counter does not yet
        /// exist, it is first created with a value of 0 (thus, the new value is -1).
        /// </summary>
        /// <param name="counter">the name of the counter</param>
```

```
        void DecrementCounter(string counter);
    }
}
```



Note

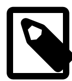
Despite being a `object` for internal compatibility, the key should be a `string` which can be converted to a `string` if it is not.

`ContextManagerUnityLoader`

In order to use context managers, one should extend a dedicated UnityLoader: `ContextManagerUnityLoader`, that will register in the unity container two instances of `ContextManager`, one for the job and one for the step. The keys used for these registrations are `BatchConstants.JobContextManagerName` and `BatchConstants.StepContextManagerName`. As usual, this class must be extended and `LoadArtifacts` must be implemented with all job artifacts registrations. If a registration should declare a ContextManager as a property, it can be done through usual unity wiring.

Example 7.21. Unity wiring example for `ContextManager`

```
container.StepScopeRegistration<IMyInterface, IMyClass>("MyKey")
    .Property("JobContextManager").Reference<IContextManager>(BatchConstants.JobContextManagerName)
    .Property("StepContextManager").Reference<IContextManager>(BatchConstants.StepContextManagerName)
    .Register();
```



Note

Unity registration can also be done through automatic injection, which is the strategy used in `AbstractExecutionListener` and `AbstractService`.

`AbstractExecutionListener` and `AbstractService`

`AbstractExecutionListener` class must be extended by the Processor classes. It supplies access to the StepContextManager and JobContextManager properties and adds a StepListener capability: in `BeforeStep` method, the job and step context managers are linked to the current job execution context and step execution context. Then, the context managers can be used in processor code. Even other services involved in processing can access these context managers by extending `AbstractService`. In any case, with these base classes, properties named `StepContextManager` and `JobContextManager` will be available and usable.

Caution

If you do not use `AbstractExecutionListener`, context managers will not be linked to correct contexts, and even if they are injected, they will remain empty shells and not work properly.

Process Adapters

ProcessAdapters are classes that enable to call a reader or a writer inside a processor class. They are useful to read several files at once: first file will be read through the reader as usual while the second file will be read during the processing. Two such classes are: `ProcessReaderAdapter` that will supply `ReadInProcess` method, while `ProcessWriterAdapter` will supply `WriteInProcess` method. Both classes have an `Adaptee` property that references the actual reader or writer, and that will be wired through Unity setup will need the StepContextManager.

Example 7.22. Example ProcessAdapter Wiring

```
…
// Process adapter
container.StepScopeRegistration<IProcessWriter<EmployeeBO>, ProcessWriterAdapter<EmployeeBO>>("AdapterKey")
    .Property("StepContextManager").Reference<IContextManager>(BatchConstants.StepContextManagerName)
    .Property("Adaptee").Reference<IItemWriter<EmployeeBO>>("WriterKey")
    .Register();

// Actual writer
container.StepScopeRegistration<IItemWriter<EmployeeBO>, FlatFileItemWriter<EmployeeBO>>("WriterKey")
    .Property("Resource").Resource("#{settings[…]}")
    …
    .Register();
…
```

Template facility

In Chapter 6, Flat File Writer was presented. Among other properties to set, for LineAggregator `Summer.Batch.Extra.Template.AbstractTemplateLineAggregator<T>` is dedicated to format the output to an external format file. The class must be extended by implementing `IEnumerable<object> GetParameters(T obj)` that converts the business object used in batch into an enumeration of values.

The template file must have a content like this: `Key:Format`, with the Format in C# fashion. Multiple lines of this form can appear with multiple keys and a format can be multi line by repeating colon.

Example 7.23. Example format file

```
EMPLOYEE: EMPID:{0} EMPNAME:{1}
```

Example 7.24. More advanced example format file

```
EMPLOYEE: EMPID:{0} EMPNAME:{1}
HEADER   :=====================================
         := Employee List for Date {0:MM/dd/yyyy} =
         :=====================================
FOOTER   :=====================================
         := End Employee List                 =
         :=====================================
```

In this example, there are three formats, and two of them span on three lines each.

The following properties are mandatory (need to be set at initialization time):

- Template : Reference to resource file containing format;
- TemplateId : key to the correct format in file.

Optional properties (Default value is provided, however can set at initialization time):

- InputEncoding : template file encoding, defaults to `Encoding.Default`;
- Culture : template file culture, default to `CultureInfo.CurrentCulture`;
- LineSeparator : line separator to be used for multi line formats, defaults to `Environment.NewLine`.

Configuring the `AbstractTemplateLineAggregator`:

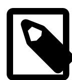
Example 7.25. Typical `AbstractTemplateLineAggregator` usage

```
…
// Writer - step1/WriteTemplateLine
container
    .StepScopeRegistration<IItemWriter<EmployeeBO>, FlatFileItemWriter<EmployeeBO>>("step1/WriteTemplateLine")
    .Property("Resource").Resource("#{settings['BA_TEMPLATE_LINE_WRITER.step1.WriteTemplateLine.FILENAME_OUT']}")
    .Property("Encoding").Value(Encoding.GetEncoding("ASCII"))
    .Property("LineAggregator")
        .Reference<ITemplateLineAggregator<EmployeeBO>>("step1/WriteTemplateLine/LineAggregator")
    .Register();

// Template Line aggregator
container.StepScopeRegistration
        <ITemplateLineAggregator<EmployeeBO>, MyTemplateLineAggregator>("step1/WriteTemplateLine/LineAggregator")
    .Property("Template")
        .Resource("#{settings['BA_TEMPLATE_LINE_WRITER.step1.WriteTemplateLine.TEMPLATE.FILENAME_IN']}")
    .Property("TemplateId").Value("EMPLOYEE")
    .Register();
…
```

With a MyTemplateLineAggregator such as:

Example 7.26. Typical TemplateLineAggregator class

```
public class MyTemplateLineAggregator : AbstractTemplateLineAggregator<EmployeeBO>
    {
        protected override IEnumerable<object> GetParameters(EmployeeBO employee)
        {
            // Compute a list of the values in EmployeeBO
            IList<object> parameters = new List<object>();
            parameters.Add(employee.Id);
            parameters.Add(employee.Name);
            return parameters;
        }
    }
```



Note

The template line aggregator will be useful with a way to switch between one template Id and another depending on the line to write. This requires a ProcessWriterAdapter, which enables to call the writer in a processor code and have a full control on its use. More details just below.

Controlling Template ID

With a simple use of the `ItemplateLineAggregator` in a writer, as explained above, the `TemplateId` is set once and for all in the Unity Loader. This means a multi-format template file would be useful only if several writers are used in the job, and each have a format, all the formats being stored in same template file. However, If a file to write must have several formats depending on the line, a more accurate control on the Template Id must be used. First, the writer with an aggregator must be declared in a process adapter, as explained above in this chapter.

Example 7.27. TemplateLineAggregator unity setup with a ProcessAdapter

```
…
// Process adapter
container.StepScopeRegistration<IProcessWriter<object>, ProcessWriterAdapter<object>>("AdapterKey")
    .Property("StepContextManager").Reference<IContextManager>(BatchConstants.StepContextManagerName)
```

```
    .Property("Adaptee").Reference<IItemWriter<EmployeeBO>>("WriterKey")
    .Register();

// Template line writer
container.StepScopeRegistration<IItemWriter<object>, FlatFileItemWriter<object>>("WriterKey")
    .Property("Resource").Resource("#{settings['…']}")
    .Property("Encoding").Value(Encoding.GetEncoding("ASCII"))
    .Property("LineAggregator").Reference<ITemplateLineAggregator<object>>("AggregatorKey")
    .Register();

// Template Line aggregator
container.StepScopeRegistration<ITemplateLineAggregator<object>, MyTemplateLineAggregator>("AggregatorKey")
    .Property("Template").Resource("#{settings['…']}")
    .Property("TemplateId").Value("EMPLOYEE")
    .Register();
…
```

With such a setup, even though a default TemplateId is assigned, it becomes possible to change it programmatically and to drive writer at will.

Example 7.28. TemplateLineAggregator usage in a process

```
…
[Dependency("AggregatorKey")]
public ITemplateLineAggregator<object> WriteTemplateLineAggregator { get; set; }

[Dependency("AdapterKey")]
public IProcessWriter<object> WriteTemplateLineWriter { get; set; }

public EmployeeBO Process(EmployeeBO employee)
{
    …
    WriteTemplateLineAggregator.TemplateId = "HEADER";
    WriteTemplateLineWriter.WriteInProcess(DateTime.Now);
    WriteTemplateLineAggregator.TemplateId = "EMPLOYEE";
    WriteTemplateLineWriter.WriteInProcess(employee);
    WriteTemplateLineAggregator.TemplateId = "FOOTER";
    WriteTemplateLineWriter.WriteInProcess(null);
    …
}
…
```

Please note that `GetParameters` method of the aggregator class must be adapted to handle different types of input objects.

Example 7.29. `GetParameters` method for heterogeneous inputs

```
…
protected override IEnumerable<object> GetParameters(object obj)
{
    IList<object> parameters = new List<object>();

    if ("HEADER".Equals(TemplateId))
    {
        parameters.Add((DateTime)obj);
    }
    else if ("EMPLOYEE".Equals(TemplateId))
    {
        var employee = (EmployeeBO)obj;
        parameters.Add(employee.Id);
        parameters.Add(employee.Name);
    }

    // Returns an empty list if FOOTER
    return parameters;
}
…
```

Appendix A. Appendix

Job File Format Xml schema

Below is the xml schema to be used to validate the Job configuration files -- see the corresponding section in this document

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema version="1.0" xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
 targetNamespace="http://www.summerbatch.com/xmlns" xmlns:sbj="http://www.summerbatch.com/xmlns">
 <xs:annotation>
  <xs:documentation> XSD for Summer batch jobs.</xs:documentation>
 </xs:annotation>
 <xs:complexType name="Job">
  <xs:sequence>
   <xs:element name="listeners" type="sbj:Listeners" minOccurs="0" maxOccurs="1"/>
   <xs:choice minOccurs="0" maxOccurs="unbounded">
    <xs:element name="flow" type="sbj:Flow" />
    <xs:element name="split" type="sbj:Split" />
    <xs:element name="step" type="sbj:Step" />
   </xs:choice>
  </xs:sequence>
  <xs:attribute name="id" use="required" type="xs:ID" />
  <xs:attribute name="restartable" use="optional" type="xs:string" />
 </xs:complexType>
 <xs:element name="job" type="sbj:Job" />
 <xs:complexType name="Listener">
  <xs:attribute name="ref" use="required" type="xs:string" />
 </xs:complexType>
 <xs:complexType name="Split">
  <xs:sequence>
   <xs:element name="flow" type="sbj:Flow" minOccurs="0" maxOccurs="unbounded" />
  </xs:sequence>
  <xs:attribute name="id" use="required" type="xs:ID" />
  <xs:attribute name="next" use="optional" type="xs:string" />
 </xs:complexType>
 <xs:complexType name="Flow">
  <xs:sequence>
   <xs:choice minOccurs="0" maxOccurs="unbounded">
    <xs:element name="flow" type="sbj:Flow" />
    <xs:element name="split" type="sbj:Split" />
    <xs:element name="step" type="sbj:Step" />
   </xs:choice>
   <xs:group ref="sbj:TransitionElements" minOccurs="0" maxOccurs="unbounded" />
  </xs:sequence>
  <xs:attribute name="id" use="required" type="xs:ID" />
  <xs:attribute name="next" use="optional" type="xs:string" />
 </xs:complexType>
 <xs:group name="TransitionElements">
  <xs:choice>
   <xs:element name="end" type="sbj:End" />
   <xs:element name="fail" type="sbj:Fail" />
   <xs:element name="next" type="sbj:Next" />
  </xs:choice>
 </xs:group>
 <xs:complexType name="Fail">
  <xs:attribute name="on" use="required" type="xs:string" />
 </xs:complexType>
 <xs:complexType name="End">
  <xs:attribute name="on" use="required" type="xs:string" />
 </xs:complexType>
 <xs:complexType name="Next">
  <xs:attribute name="on" use="required" type="xs:string" />
  <xs:attribute name="to" use="required" type="xs:string" />
 </xs:complexType>
 <xs:complexType name="Step">
  <xs:sequence>
   <xs:element name="listeners" type="sbj:Listeners" minOccurs="0" maxOccurs="1"/>
   <xs:choice minOccurs="0" maxOccurs="1">
    <xs:element name="batchlet" type="sbj:Batchlet" />
    <xs:element name="chunk" type="sbj:Chunk" />
   </xs:choice>
```

```
    <xs:element name="partition" type="sbj:Partition" minOccurs="0" maxOccurs="1" />
    <xs:group ref="sbj:TransitionElements" minOccurs="0" maxOccurs="unbounded" />
   </xs:sequence>
   <xs:attribute name="id" use="required" type="xs:ID" />
   <xs:attribute name="next" use="optional" type="xs:string" />
  </xs:complexType>
  <xs:complexType name="Batchlet">
   <xs:attribute name="ref" use="required" type="xs:string" />
  </xs:complexType>
  <xs:complexType name="Chunk">
   <xs:sequence>
    <xs:element name="reader" type="sbj:ItemReader" />
    <xs:element name="processor" type="sbj:ItemProcessor" minOccurs="0" maxOccurs="1" />
    <xs:element name="writer" type="sbj:ItemWriter" />
   </xs:sequence>
   <xs:attribute name="item-count" use="optional" type="xs:string" />
  </xs:complexType>
  <xs:complexType name="ItemReader">
   <xs:attribute name="ref" use="required" type="xs:string" />
  </xs:complexType>
  <xs:complexType name="ItemProcessor">
   <xs:attribute name="ref" use="required" type="xs:string" />
  </xs:complexType>
  <xs:complexType name="ItemWriter">
   <xs:attribute name="ref" use="required" type="xs:string" />
  </xs:complexType>
  <xs:complexType name="Listeners">
   <xs:sequence>
    <xs:element name="listener" type="sbj:Listener" minOccurs="0" maxOccurs="unbounded" />
   </xs:sequence>
  </xs:complexType>
  <xs:complexType name="Partition">
   <xs:sequence>
    <xs:element name="mapper" type="sbj:PartitionMapper" />
   </xs:sequence>
  </xs:complexType>
  <xs:complexType name="PartitionMapper">
   <xs:attribute name="ref" use="required" type="xs:string" />
   <xs:attribute name="grid-size" use="optional" type="xs:int" />
  </xs:complexType>
</xs:schema>
```

Ebcdic File Format Xml schema

Below is the xml schema to be used to validate the copybook xml files (used by EbcdicFileReader and EbcdicFileWriter -- see the corresponding section in this document)

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="FileFormat">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="RecordFormat" type="RecordFormat" maxOccurs="unbounded"/>
   </xsd:sequence>
   <xsd:attribute name="ConversionTable" type="xsd:string"/>
   <xsd:attribute name="distinguishFieldSize" type="xsd:integer"/>
   <xsd:attribute name="newLineSize" type="xsd:integer"/>
   <xsd:attribute name="dataFileImplementation" type="xsd:string"/>
   <xsd:attribute name="headerSize" type="xsd:integer"/>
  </xsd:complexType>
 </xsd:element>
 <xsd:complexType name="RecordFormat">
  <xsd:complexContent>
   <xsd:extension base="FieldsList">
    <xsd:attribute name="distinguishFieldValue" type="xsd:string"/>
    <xsd:attribute name="cobolRecordName" type="xsd:string"/>
   </xsd:extension>
  </xsd:complexContent>
 </xsd:complexType>
 <xsd:complexType name="FieldsGroup">
  <xsd:complexContent>
   <xsd:extension base="FieldsList">
```

```
    <xsd:attribute name="Name" type="xsd:string"/>
    <xsd:attribute name="Occurs" type="xsd:integer"/>
    <xsd:attribute name="DependingOn" type="xsd:string"/>
    <xsd:attribute name="Redefined" type="xsd:boolean"/>
    <xsd:attribute name="Redefines" type="xsd:string"/>
   </xsd:extension>
  </xsd:complexContent>
 </xsd:complexType>
 <xsd:complexType name="FieldsList">
  <xsd:choice maxOccurs="unbounded">
   <xsd:element name="FieldFormat" type="FieldFormat" minOccurs="0" maxOccurs="unbounded"/>
   <xsd:element name="FieldsGroup" type="FieldsGroup" minOccurs="0" maxOccurs="unbounded"/>
  </xsd:choice>
 </xsd:complexType>
 <xsd:complexType name="FieldFormat">
  <xsd:attribute name="Name" type="xsd:string"/>
  <xsd:attribute name="Occurs" type="xsd:integer"/>
  <xsd:attribute name="DependingOn" type="xsd:string"/>
  <xsd:attribute name="Redefined" type="xsd:boolean"/>
  <xsd:attribute name="Redefines" type="xsd:string"/>
  <xsd:attribute name="Size" type="xsd:integer"/>
  <xsd:attribute name="Type" type="xsd:string"/>
  <xsd:attribute name="Decimal" type="xsd:integer"/>
  <xsd:attribute name="Signed" type="xsd:boolean"/>
  <xsd:attribute name="ImpliedDecimal" type="xsd:boolean"/>
  <xsd:attribute name="Value" type="xsd:string"/>
  <xsd:attribute name="Picture" type="xsd:string"/>
 </xsd:complexType>
</xsd:schema>
```

Database Repository scripts per vendor

Sql Server scripts

Repository creation script

```
CREATE TABLE BATCH_JOB_INSTANCE  (
 JOB_INSTANCE_ID BIGINT  NOT NULL PRIMARY KEY ,
 VERSION BIGINT NULL,
 JOB_NAME VARCHAR(100) NOT NULL,
 JOB_KEY VARCHAR(32) NOT NULL,
 constraint JOB_INST_UN unique (JOB_NAME, JOB_KEY)
) ;

CREATE TABLE BATCH_JOB_EXECUTION  (
 JOB_EXECUTION_ID BIGINT  NOT NULL PRIMARY KEY ,
 VERSION BIGINT NULL,
 JOB_INSTANCE_ID BIGINT NOT NULL,
 CREATE_TIME DATETIME NOT NULL,
 START_TIME DATETIME DEFAULT NULL ,
 END_TIME DATETIME DEFAULT NULL ,
 STATUS VARCHAR(10) NULL,
 EXIT_CODE VARCHAR(2500) NULL,
 EXIT_MESSAGE VARCHAR(2500) NULL,
 LAST_UPDATED DATETIME NULL,
 JOB_CONFIGURATION_LOCATION VARCHAR(2500) NULL,
 constraint JOB_INST_EXEC_FK foreign key (JOB_INSTANCE_ID)
 references BATCH_JOB_INSTANCE(JOB_INSTANCE_ID)
) ;

CREATE TABLE BATCH_JOB_EXECUTION_PARAMS  (
 JOB_EXECUTION_ID BIGINT NOT NULL ,
 TYPE_CD VARCHAR(6) NOT NULL ,
 KEY_NAME VARCHAR(100) NOT NULL ,
 STRING_VAL VARCHAR(250) NULL,
 DATE_VAL DATETIME DEFAULT NULL ,
 LONG_VAL BIGINT NULL,
 DOUBLE_VAL DOUBLE PRECISION NULL,
 IDENTIFYING CHAR(1) NOT NULL ,
 constraint JOB_EXEC_PARAMS_FK foreign key (JOB_EXECUTION_ID)
 references BATCH_JOB_EXECUTION(JOB_EXECUTION_ID)
```

```
) ;

CREATE TABLE BATCH_STEP_EXECUTION  (
 STEP_EXECUTION_ID BIGINT  NOT NULL PRIMARY KEY ,
 VERSION BIGINT NOT NULL,
 STEP_NAME VARCHAR(100) NOT NULL,
 JOB_EXECUTION_ID BIGINT NOT NULL,
 START_TIME DATETIME NOT NULL ,
 END_TIME DATETIME DEFAULT NULL ,
 STATUS VARCHAR(10) NULL,
 COMMIT_COUNT BIGINT NULL,
 READ_COUNT BIGINT NULL,
 FILTER_COUNT BIGINT NULL,
 WRITE_COUNT BIGINT NULL,
 READ_SKIP_COUNT BIGINT NULL,
 WRITE_SKIP_COUNT BIGINT NULL,
 PROCESS_SKIP_COUNT BIGINT NULL,
 ROLLBACK_COUNT BIGINT NULL,
 EXIT_CODE VARCHAR(2500) NULL,
 EXIT_MESSAGE VARCHAR(2500) NULL,
 LAST_UPDATED DATETIME NULL,
 constraint JOB_EXEC_STEP_FK foreign key (JOB_EXECUTION_ID)
 references BATCH_JOB_EXECUTION(JOB_EXECUTION_ID)
) ;

CREATE TABLE BATCH_STEP_EXECUTION_CONTEXT  (
 STEP_EXECUTION_ID BIGINT NOT NULL PRIMARY KEY,
 SERIALIZED_CONTEXT VARBINARY(MAX) NULL,
 constraint STEP_EXEC_CTX_FK foreign key (STEP_EXECUTION_ID)
 references BATCH_STEP_EXECUTION(STEP_EXECUTION_ID)
) ;

CREATE TABLE BATCH_JOB_EXECUTION_CONTEXT  (
 JOB_EXECUTION_ID BIGINT NOT NULL PRIMARY KEY,
 SERIALIZED_CONTEXT VARBINARY(MAX) NULL,
 constraint JOB_EXEC_CTX_FK foreign key (JOB_EXECUTION_ID)
 references BATCH_JOB_EXECUTION(JOB_EXECUTION_ID)
) ;

CREATE TABLE BATCH_STEP_EXECUTION_SEQ (ID BIGINT IDENTITY);
CREATE TABLE BATCH_JOB_EXECUTION_SEQ (ID BIGINT IDENTITY);
CREATE TABLE BATCH_JOB_SEQ (ID BIGINT IDENTITY);
```

Repository drop script

```
DROP TABLE  BATCH_STEP_EXECUTION_CONTEXT ;
DROP TABLE  BATCH_JOB_EXECUTION_CONTEXT ;
DROP TABLE  BATCH_STEP_EXECUTION ;
DROP TABLE  BATCH_JOB_EXECUTION_PARAMS ;
DROP TABLE  BATCH_JOB_EXECUTION ;
DROP TABLE  BATCH_JOB_INSTANCE ;

DROP TABLE  BATCH_STEP_EXECUTION_SEQ ;
DROP TABLE  BATCH_JOB_EXECUTION_SEQ ;
DROP TABLE  BATCH_JOB_SEQ ;
```

Oracle scripts

Repository creation script

```
CREATE TABLE BATCH_JOB_INSTANCE  (
 JOB_INSTANCE_ID NUMBER(18,0)  NOT NULL PRIMARY KEY ,
 VERSION NUMBER(18,0) ,
 JOB_NAME VARCHAR2(100) NOT NULL,
 JOB_KEY VARCHAR2(32) NOT NULL,
 constraint JOB_INST_UN unique (JOB_NAME, JOB_KEY)
) ;

CREATE TABLE BATCH_JOB_EXECUTION  (
 JOB_EXECUTION_ID NUMBER(18,0)  NOT NULL PRIMARY KEY ,
 VERSION NUMBER(18,0)  ,
 JOB_INSTANCE_ID NUMBER(18,0) NOT NULL,
```

```
 CREATE_TIME TIMESTAMP NOT NULL,
 START_TIME TIMESTAMP DEFAULT NULL ,
 END_TIME TIMESTAMP DEFAULT NULL ,
 STATUS VARCHAR2(10) ,
 EXIT_CODE VARCHAR2(2500) ,
 EXIT_MESSAGE VARCHAR2(2500) ,
 LAST_UPDATED TIMESTAMP,
 JOB_CONFIGURATION_LOCATION VARCHAR(2500) NULL,
 constraint JOB_INST_EXEC_FK foreign key (JOB_INSTANCE_ID)
 references BATCH_JOB_INSTANCE(JOB_INSTANCE_ID)
) ;

CREATE TABLE BATCH_JOB_EXECUTION_PARAMS  (
 JOB_EXECUTION_ID NUMBER(18,0) NOT NULL ,
 TYPE_CD VARCHAR2(6) NOT NULL ,
 KEY_NAME VARCHAR2(100) NOT NULL ,
 STRING_VAL VARCHAR2(250) ,
 DATE_VAL TIMESTAMP DEFAULT NULL ,
 LONG_VAL NUMBER(18,0) ,
 DOUBLE_VAL BINARY_DOUBLE ,
 IDENTIFYING CHAR(1) NOT NULL ,
 constraint JOB_EXEC_PARAMS_FK foreign key (JOB_EXECUTION_ID)
 references BATCH_JOB_EXECUTION(JOB_EXECUTION_ID)
) ;

CREATE TABLE BATCH_STEP_EXECUTION  (
 STEP_EXECUTION_ID NUMBER(18,0)  NOT NULL PRIMARY KEY ,
 VERSION NUMBER(18,0) NOT NULL,
 STEP_NAME VARCHAR2(100) NOT NULL,
 JOB_EXECUTION_ID NUMBER(18,0) NOT NULL,
 START_TIME TIMESTAMP NOT NULL ,
 END_TIME TIMESTAMP DEFAULT NULL ,
 STATUS VARCHAR2(10) ,
 COMMIT_COUNT NUMBER(18,0) ,
 READ_COUNT NUMBER(18,0) ,
 FILTER_COUNT NUMBER(18,0) ,
 WRITE_COUNT NUMBER(18,0) ,
 READ_SKIP_COUNT NUMBER(18,0) ,
 WRITE_SKIP_COUNT NUMBER(18,0) ,
 PROCESS_SKIP_COUNT NUMBER(18,0) ,
 ROLLBACK_COUNT NUMBER(18,0) ,
 EXIT_CODE VARCHAR2(2500) ,
 EXIT_MESSAGE VARCHAR2(2500) ,
 LAST_UPDATED TIMESTAMP,
 constraint JOB_EXEC_STEP_FK foreign key (JOB_EXECUTION_ID)
 references BATCH_JOB_EXECUTION(JOB_EXECUTION_ID)
) ;

CREATE TABLE BATCH_STEP_EXECUTION_CONTEXT  (
 STEP_EXECUTION_ID NUMBER(18,0) NOT NULL PRIMARY KEY,
 SHORT_CONTEXT VARCHAR2(2500) NOT NULL,
 SERIALIZED_CONTEXT CLOB ,
 constraint STEP_EXEC_CTX_FK foreign key (STEP_EXECUTION_ID)
 references BATCH_STEP_EXECUTION(STEP_EXECUTION_ID)
) ;

CREATE TABLE BATCH_JOB_EXECUTION_CONTEXT  (
 JOB_EXECUTION_ID NUMBER(18,0) NOT NULL PRIMARY KEY,
 SHORT_CONTEXT VARCHAR2(2500) NOT NULL,
 SERIALIZED_CONTEXT CLOB ,
 constraint JOB_EXEC_CTX_FK foreign key (JOB_EXECUTION_ID)
 references BATCH_JOB_EXECUTION(JOB_EXECUTION_ID)
) ;

CREATE SEQUENCE BATCH_STEP_EXECUTION_SEQ START WITH 0 MINVALUE 0
 MAXVALUE 9223372036854775807 NOCYCLE;
CREATE SEQUENCE BATCH_JOB_EXECUTION_SEQ START WITH 0 MINVALUE 0
 MAXVALUE 9223372036854775807 NOCYCLE;
CREATE SEQUENCE BATCH_JOB_SEQ START WITH 0 MINVALUE 0 MAXVALUE 9223372036854775807 NOCYCLE;
```

Repository drop script

```
DROP TABLE  BATCH_STEP_EXECUTION_CONTEXT ;
DROP TABLE  BATCH_JOB_EXECUTION_CONTEXT ;
```

```
DROP TABLE  BATCH_STEP_EXECUTION ;
DROP TABLE  BATCH_JOB_EXECUTION_PARAMS ;
DROP TABLE  BATCH_JOB_EXECUTION ;
DROP TABLE  BATCH_JOB_INSTANCE ;

DROP SEQUENCE  BATCH_STEP_EXECUTION_SEQ ;
DROP SEQUENCE  BATCH_JOB_EXECUTION_SEQ ;
DROP SEQUENCE  BATCH_JOB_SEQ ;
```

IBM DB2 scripts

Repository creation script

```
CREATE TABLE BATCH_JOB_INSTANCE  (
 JOB_INSTANCE_ID BIGINT  NOT NULL PRIMARY KEY ,
 VERSION BIGINT ,
 JOB_NAME VARCHAR(100) NOT NULL,
 JOB_KEY VARCHAR(32) NOT NULL,
 constraint JOB_INST_UN unique (JOB_NAME, JOB_KEY)
) ;

CREATE TABLE BATCH_JOB_EXECUTION  (
 JOB_EXECUTION_ID BIGINT  NOT NULL PRIMARY KEY ,
 VERSION BIGINT  ,
 JOB_INSTANCE_ID BIGINT NOT NULL,
 CREATE_TIME TIMESTAMP NOT NULL,
 START_TIME TIMESTAMP DEFAULT NULL ,
 END_TIME TIMESTAMP DEFAULT NULL ,
 STATUS VARCHAR(10) ,
 EXIT_CODE VARCHAR(2500) ,
 EXIT_MESSAGE VARCHAR(2500) ,
 LAST_UPDATED TIMESTAMP,
 JOB_CONFIGURATION_LOCATION VARCHAR(2500) ,
 constraint JOB_INST_EXEC_FK foreign key (JOB_INSTANCE_ID)
 references BATCH_JOB_INSTANCE(JOB_INSTANCE_ID)
) ;

CREATE TABLE BATCH_JOB_EXECUTION_PARAMS  (
 JOB_EXECUTION_ID BIGINT NOT NULL ,
 TYPE_CD VARCHAR(6) NOT NULL ,
 KEY_NAME VARCHAR(100) NOT NULL ,
 STRING_VAL VARCHAR(250) ,
 DATE_VAL TIMESTAMP DEFAULT NULL ,
 LONG_VAL BIGINT ,
 DOUBLE_VAL DOUBLE PRECISION ,
 IDENTIFYING CHAR(1) NOT NULL ,
 constraint JOB_EXEC_PARAMS_FK foreign key (JOB_EXECUTION_ID)
 references BATCH_JOB_EXECUTION(JOB_EXECUTION_ID)
) ;

CREATE TABLE BATCH_STEP_EXECUTION  (
 STEP_EXECUTION_ID BIGINT  NOT NULL PRIMARY KEY ,
 VERSION BIGINT NOT NULL,
 STEP_NAME VARCHAR(100) NOT NULL,
 JOB_EXECUTION_ID BIGINT NOT NULL,
 START_TIME TIMESTAMP NOT NULL ,
 END_TIME TIMESTAMP DEFAULT NULL ,
 STATUS VARCHAR(10) ,
 COMMIT_COUNT BIGINT ,
 READ_COUNT BIGINT ,
 FILTER_COUNT BIGINT ,
 WRITE_COUNT BIGINT ,
 READ_SKIP_COUNT BIGINT ,
 WRITE_SKIP_COUNT BIGINT ,
 PROCESS_SKIP_COUNT BIGINT ,
 ROLLBACK_COUNT BIGINT ,
 EXIT_CODE VARCHAR(2500) ,
 EXIT_MESSAGE VARCHAR(2500) ,
 LAST_UPDATED TIMESTAMP,
 constraint JOB_EXEC_STEP_FK foreign key (JOB_EXECUTION_ID)
 references BATCH_JOB_EXECUTION(JOB_EXECUTION_ID)
) ;

CREATE TABLE BATCH_STEP_EXECUTION_CONTEXT  (
```

```
 STEP_EXECUTION_ID BIGINT NOT NULL PRIMARY KEY,
 SERIALIZED_CONTEXT BLOB ,
 constraint STEP_EXEC_CTX_FK foreign key (STEP_EXECUTION_ID)
 references BATCH_STEP_EXECUTION(STEP_EXECUTION_ID)
) ;

CREATE TABLE BATCH_JOB_EXECUTION_CONTEXT  (
 JOB_EXECUTION_ID BIGINT NOT NULL PRIMARY KEY,
 SERIALIZED_CONTEXT BLOB ,
 constraint JOB_EXEC_CTX_FK foreign key (JOB_EXECUTION_ID)
 references BATCH_JOB_EXECUTION(JOB_EXECUTION_ID)
) ;

CREATE SEQUENCE BATCH_STEP_EXECUTION_SEQ AS BIGINT MAXVALUE 9223372036854775807 NO CYCLE;
CREATE SEQUENCE BATCH_JOB_EXECUTION_SEQ AS BIGINT MAXVALUE 9223372036854775807 NO CYCLE;
CREATE SEQUENCE BATCH_JOB_SEQ AS BIGINT MAXVALUE 9223372036854775807 NO CYCLE;
```

Repository drop script

```
DROP TABLE  BATCH_STEP_EXECUTION_CONTEXT ;
DROP TABLE  BATCH_JOB_EXECUTION_CONTEXT ;
DROP TABLE  BATCH_STEP_EXECUTION ;
DROP TABLE  BATCH_JOB_EXECUTION_PARAMS ;
DROP TABLE  BATCH_JOB_EXECUTION ;
DROP TABLE  BATCH_JOB_INSTANCE ;

DROP SEQUENCE  BATCH_STEP_EXECUTION_SEQ ;
DROP SEQUENCE  BATCH_JOB_EXECUTION_SEQ ;
DROP SEQUENCE  BATCH_JOB_SEQ ;
```

NOT OFFICIALLY SUPPORTED : PostgreSQL scripts

Repository creation script

```
CREATE TABLE BATCH_JOB_INSTANCE  (
 JOB_INSTANCE_ID BIGINT  NOT NULL PRIMARY KEY ,
 VERSION BIGINT ,
 JOB_NAME VARCHAR(100) NOT NULL,
 JOB_KEY VARCHAR(32) NOT NULL,
 constraint JOB_INST_UN unique (JOB_NAME, JOB_KEY)
) ;

CREATE TABLE BATCH_JOB_EXECUTION  (
 JOB_EXECUTION_ID BIGINT  NOT NULL PRIMARY KEY ,
 VERSION BIGINT  ,
 JOB_INSTANCE_ID BIGINT NOT NULL,
 CREATE_TIME TIMESTAMP NOT NULL,
 START_TIME TIMESTAMP DEFAULT NULL ,
 END_TIME TIMESTAMP DEFAULT NULL ,
 STATUS VARCHAR(10) ,
 EXIT_CODE VARCHAR(2500) ,
 EXIT_MESSAGE VARCHAR(2500) ,
 LAST_UPDATED TIMESTAMP,
 JOB_CONFIGURATION_LOCATION VARCHAR(2500) NULL,
 constraint JOB_INST_EXEC_FK foreign key (JOB_INSTANCE_ID)
 references BATCH_JOB_INSTANCE(JOB_INSTANCE_ID)
) ;

CREATE TABLE BATCH_JOB_EXECUTION_PARAMS  (
 JOB_EXECUTION_ID BIGINT NOT NULL ,
 TYPE_CD VARCHAR(6) NOT NULL ,
 KEY_NAME VARCHAR(100) NOT NULL ,
 STRING_VAL VARCHAR(250) ,
 DATE_VAL TIMESTAMP DEFAULT NULL ,
 LONG_VAL BIGINT ,
 DOUBLE_VAL DOUBLE PRECISION ,
 IDENTIFYING CHAR(1) NOT NULL ,
 constraint JOB_EXEC_PARAMS_FK foreign key (JOB_EXECUTION_ID)
 references BATCH_JOB_EXECUTION(JOB_EXECUTION_ID)
) ;

CREATE TABLE BATCH_STEP_EXECUTION  (
```

```
 STEP_EXECUTION_ID BIGINT  NOT NULL PRIMARY KEY ,
 VERSION BIGINT NOT NULL,
 STEP_NAME VARCHAR(100) NOT NULL,
 JOB_EXECUTION_ID BIGINT NOT NULL,
 START_TIME TIMESTAMP NOT NULL ,
 END_TIME TIMESTAMP DEFAULT NULL ,
 STATUS VARCHAR(10) ,
 COMMIT_COUNT BIGINT ,
 READ_COUNT BIGINT ,
 FILTER_COUNT BIGINT ,
 WRITE_COUNT BIGINT ,
 READ_SKIP_COUNT BIGINT ,
 WRITE_SKIP_COUNT BIGINT ,
 PROCESS_SKIP_COUNT BIGINT ,
 ROLLBACK_COUNT BIGINT ,
 EXIT_CODE VARCHAR(2500) ,
 EXIT_MESSAGE VARCHAR(2500) ,
 LAST_UPDATED TIMESTAMP,
 constraint JOB_EXEC_STEP_FK foreign key (JOB_EXECUTION_ID)
 references BATCH_JOB_EXECUTION(JOB_EXECUTION_ID)
) ;

CREATE TABLE BATCH_STEP_EXECUTION_CONTEXT  (
 STEP_EXECUTION_ID BIGINT NOT NULL PRIMARY KEY,
 SERIALIZED_CONTEXT BYTEA ,
 constraint STEP_EXEC_CTX_FK foreign key (STEP_EXECUTION_ID)
 references BATCH_STEP_EXECUTION(STEP_EXECUTION_ID)
) ;

CREATE TABLE BATCH_JOB_EXECUTION_CONTEXT  (
 JOB_EXECUTION_ID BIGINT NOT NULL PRIMARY KEY,
 SERIALIZED_CONTEXT BYTEA ,
 constraint JOB_EXEC_CTX_FK foreign key (JOB_EXECUTION_ID)
 references BATCH_JOB_EXECUTION(JOB_EXECUTION_ID)
) ;

CREATE SEQUENCE BATCH_STEP_EXECUTION_SEQ MAXVALUE 9223372036854775807 NO CYCLE;
CREATE SEQUENCE BATCH_JOB_EXECUTION_SEQ MAXVALUE 9223372036854775807 NO CYCLE;
CREATE SEQUENCE BATCH_JOB_SEQ MAXVALUE 9223372036854775807 NO CYCLE;
```

Repository drop script

```
DROP TABLE  BATCH_STEP_EXECUTION_CONTEXT ;
DROP TABLE  BATCH_JOB_EXECUTION_CONTEXT ;
DROP TABLE  BATCH_STEP_EXECUTION ;
DROP TABLE  BATCH_JOB_EXECUTION_PARAMS ;
DROP TABLE  BATCH_JOB_EXECUTION ;
DROP TABLE  BATCH_JOB_INSTANCE ;

DROP SEQUENCE  BATCH_STEP_EXECUTION_SEQ ;
DROP SEQUENCE  BATCH_JOB_EXECUTION_SEQ ;
DROP SEQUENCE  BATCH_JOB_SEQ ;
```

Index

A
AbstractTemplateLineAggregator, 56
Adapters, 55
AssertUpdate, 36

B
Batch Control Flow, 15

C
ConfigurationManager, 35
ConnectionStringSettings, 32, 35
Context, 53

D
DatabaseBatchItemWriter, 35
DataReaderItemReader, 32
DateParser, 27
DbParameterSourceProvider, 35
DbProviderFactory, 32, 35
DefaultFieldSet, 25
DefaultLineMapper, 25
DelimitedLineAggregator, 30
DelimitedLineTokenizer, 25
DFSORT, 50
DictionaryParameterSource, 33

E
EbcdicFileReader, 40
EbcdicFileWriter, 44
EmailTasklet, 47
EmptyFileCheckTasklet, 49
Encoding, 28

F
FixedLengthTokenizer, 25
FlatFileItemReader, 25
FlatFileItemWriter, 28
FormatterLineAggregator, 30, 31
FtpGetTasklet, 46
FtpPutTasklet, 45

G
Generation Data Groups, 52

I
IBM DB2 support, 38
IDatabaseExtension, 38
IDataFieldMaxValueIncrementer, 38
IFieldExtractor, 31
IFieldSet
  DefaultFieldSet, 25
IFieldSetMapper, 25, 27
IItemProcessor<TIn, TOut>, 18
IItemReader<T>, 18
IItemWriter<T>, 18
IJobExecutionListener, 13, 18
ILineAggregator, 29, 30
ILineMapper
  DefaultLineMapper, 25
ILineTokenizer
  FixedLengthTokenizer
    DelimitedLineTokenizer, 25
IListableJobLocator, 18
Include Card, 51
Inrec Card, 52
IPlaceHolderGetter, 38
IQueryParameterSource, 32, 35
IQueryParameterSourceProvider, 35
IStepExecutionListener, 18
ITaskExecutor, 18
ITasklet, 18

J
Job Launcher, 18
Job Listeners, 12
Job Specification Language (JSL), 12
  Flow, 14
  Job, 12
  Split, 15
  Step, 13
Job xml Configuration, 13, 14, 14, 15, 15, 15, 16, 16
Job XML Configuration, 26, 29, 34, 37, 43, 44, 46, 47, 48, 49, 50

M
Microsoft SQL Server support, 37

O
Omit Card, 51
Oracle Database support, 38
Outrec Card, 52

P
Parameter Source, 32
PassThroughFieldExtractor, 32
PassThroughLineAggregator, 30
PostgreSQL additional database support example, 38
PropertyFieldExtractor, 31
PropertyParameterSource, 33, 36
PropertyParameterSourceProvider, 36, 37

Q
Query, 32, 35
query parameters, 32, 35

R
Restartability, 12

RowMapper, 32, 34

S

Scopes, 19
    Singleton Scope, 19
    Step Scope, 19
SimpleAsyncTaskExecutor, 18
Sort Card, 51
Sort Tasklet, 50
SqlScriptRunnerTasklet, 49
SyncTaskExecutor, 18

T

Task Executor, 18
Template, 56

U

Unity : Injection Member Methods, 21
Unity : Injection Parameter Value Methods, 21
Unity Configuration, 17, 20, 27, 31, 34, 37, 43, 44, 46, 47, 49, 50, 56, 57, 57

X

XSL Configuration, 12