# Summer Batch - Reference Guide


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

# Summer Batch - Reference Guide

by The Summer Batch team

1.0.0 RELEASE - November 2015

# Table of Contents

# List of Figures

# List of Examples

# Chapter 1. Introduction

## Summer Batch Introduction

The idea behind Summer Batch is to bring to the .NET community an efficient, open-source, lightweight batch framework, taking full advantages of the C# platform and aimed at fulfilling typical enterprise bulk processing needs. The well known JSR-352 -- despite being java colored -- is the specification we chose to support, at least *partially*.

Our primary goals are to help users to :

- migrate smoothly their batch legacy from mainframe to modern Microsoft®-based environments;
- build new batch solutions to be run on 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 the very essentials parts of the JSR-352 specification, covering the needs for the large majority of users, in particular in the scope of legacy batch modernization.

As a complement, 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

- Do you need to treat large sets of data, with a typical read-process-write repetitive scenario ?
- Does this bulk processing has to be non-interactive, highly efficient and scalable ?
- Is C# your favorite development and runtime platform ?

If you answered yes to these three questions, then Summer Batch is the framework you should consider for your batch development.

Summer Batch provides the notable following 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 the 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);

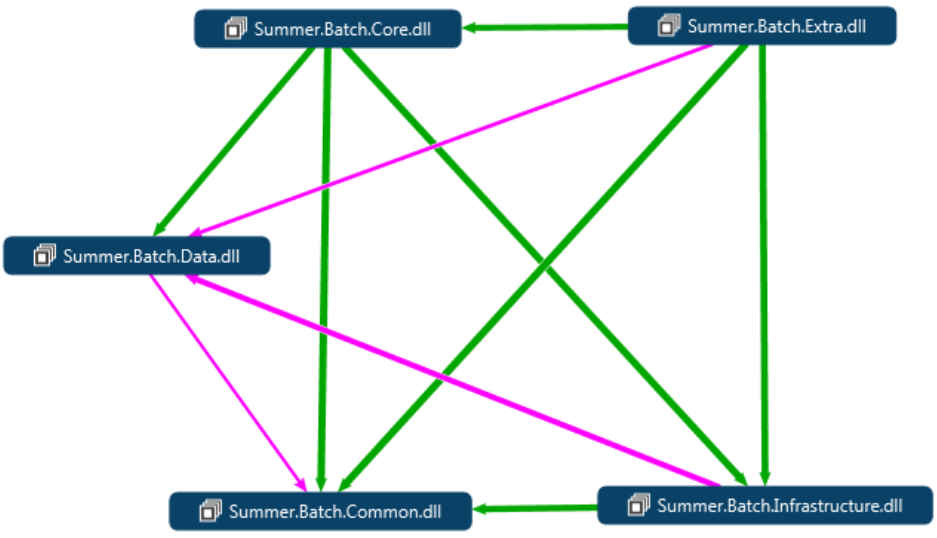
# Architecture overview

The Summer Batch solution is split into 5 components:

- Summer Batch Common : shared artefacts, 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 artefacts;
- Summer Batch Core : this is the main component that contains the batch engine, the job repository, launchers, listeners, etc ...;
- Summer Batch Extra : additional readers/writers (EBCDIC), dedicated batchlets implementations (FTP support, Email support, ...) and additional framework facilities;

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

**Figure 1.1. Summer Batch dependencies analysis**




# Programming guidelines

We believe that the key objectives of batch programming are, in that order : reliability, performance and maintenability.

Performance is dependant on a large set of factors, but some strong lines should be kept in mind :

- Shorten the path between the data and the data processor, whenever possible; Reducing the network overhead for example can have a significant positive impact on global batch performance when dealing with databases.
- For critical performance needs, using the parallel processing facility can be the way to go, but generally involves some extra work to be done (smart data partitioning).
- Carefully specify the 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 pertubations if the data are being consumed by other appliances, in case of a batch failure (since the rollback will deal with a large set of data).
- Hunt for unnecessary operations in the code; In particular, be careful with logger calls that can seriously degrade overall performance when used on a fine granularity basis.
- Follow general SQL statement performance guidelines. In particular -- a very non exclusive list ... --:
  - Make sure the SQL queries are performing well, and that all databases indexes have been created;
  - Pay attention to the order of restrictions (in the WHERE clause) by inspecting the execution plan, to ensure the most discriminant filters are invoked first.
  - Hunt for unnecessary outer joins.
  - Only select the fields that are really needed by processing.
- Keep the I/O consumption profile low;

Regarding maintenability and reliability:
Beyond general C# programming guidelines, the main advice is : keep it simple. In particular, having some intricate steps sequences with multiple ways within a given job make things hard to maintain and harder to test.

## Important

This reference guide makes the assumption that you know your way around Visual Studio and creating projects and solutions. However, to help users set up their first Summer Batch project, we wrote the Getting Started guide. We believe that reading this tutorial should be done prior to digging into this reference documentation.

# Chapter 2. Supported Features

## The JSR 352 as starting point

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

Hence, Summer Batch uses the main architecture carried by this specification, and most of its features are provided. 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 legacy Cobol batches to the Microsoft® .NET world.

## Main features

These features are widely supported in JSR-352 batch frameworks

- 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 the 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 the readers and writers and your domain classes
- Batch contexts at step level and job level
- XML design for the main batch architecture, C# design for the 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. The batch vocabulary and mechanisms

## The 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 the box C# implementations and a great extensibility that allows users to supply their own.

Batch developers ought to understand the vocabulary explained below, as it describes these key concepts and is heavily reused throughout this documentation. Anyway, it is the exact same vocabulary as in any batch development process in any language, so it should be familiar to most.

## 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**

```
<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 the 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 the XML document as a simple reference to a class, that performs the task as a whole. In the previous example the job was a sequence of two simple batchlets.

## Chunk handling

Most of the time, a step is a read-process-write task, and the 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, the current chunk will be rollbacked, while all the previous processing will be saved. And on restart (if restart was enabled, which requires a database-persisted repository), the job will be restarted at the exact chunk where the failure happened.

In the following example, the step will be processed through groups of 1000 records read by the 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 the step phase that retrieves data from a given source (database, file, etc.). It supplies items from the source until no more are available, in which case it will return null, and its processing is complete.

# Item Processor

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

In case no processor is supplied, the read data is transmitted as is to the 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 the rollback mechanics explained above.

# Repository

The Job repository is where the elements stated above are persisted. It can be in memory or in database, and it must be in database if the 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 the repository, start or restart a job, query for existing jobs, etc. The BatchRuntime is the entry point of any Summer Batch program. It is a static factory returning a fully setup Job Operator with the correct job definition. It enables to get an operator that is ready to start the job.

# The batch mechanisms

Most of the 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 behaviour

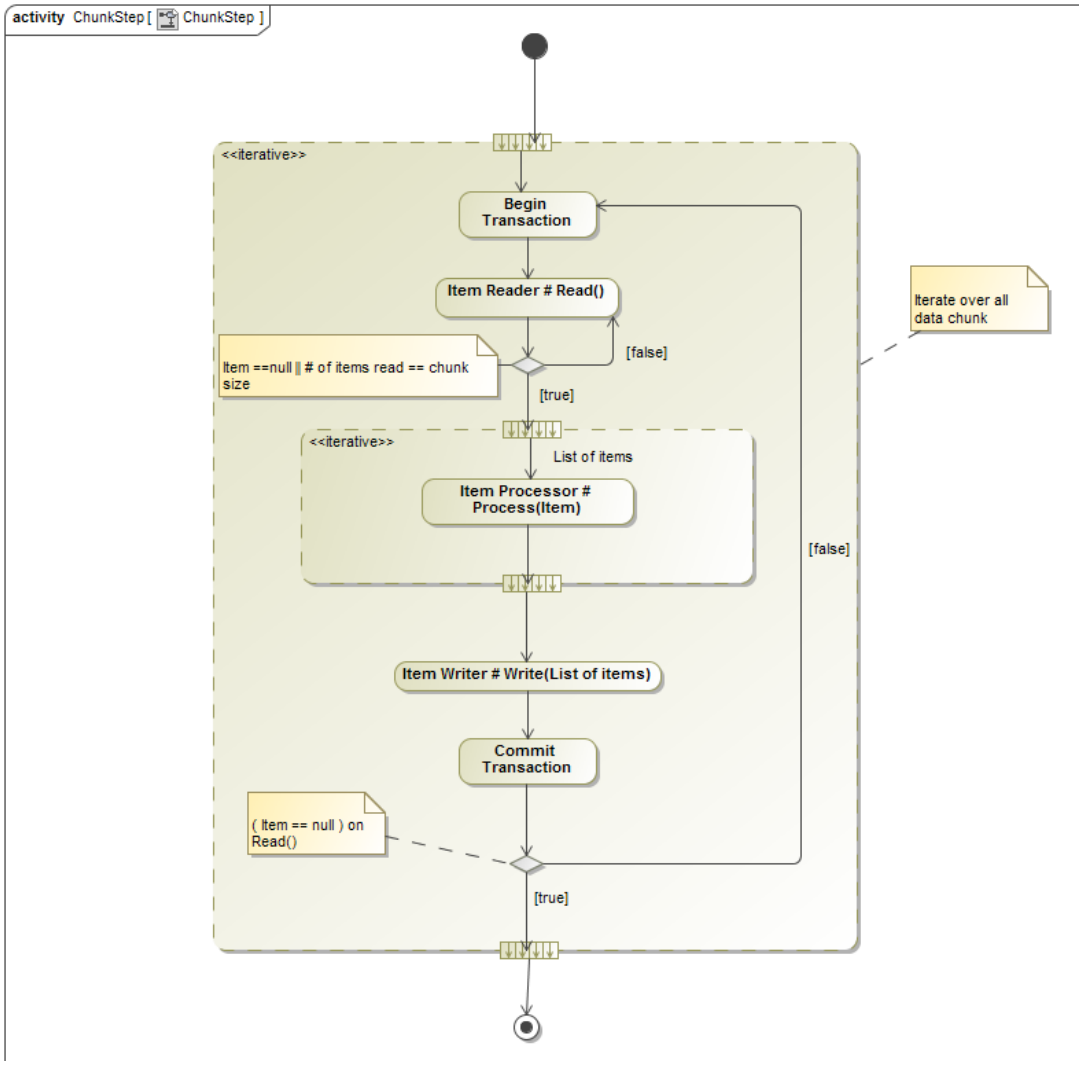
A chunk oriented step is made of -- in that order -- :

- An Item Reader
- An optional Item Processor
- An Item Writer

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

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

**Figure 3.1. Chunk oriented step : basic transactional behaviour**



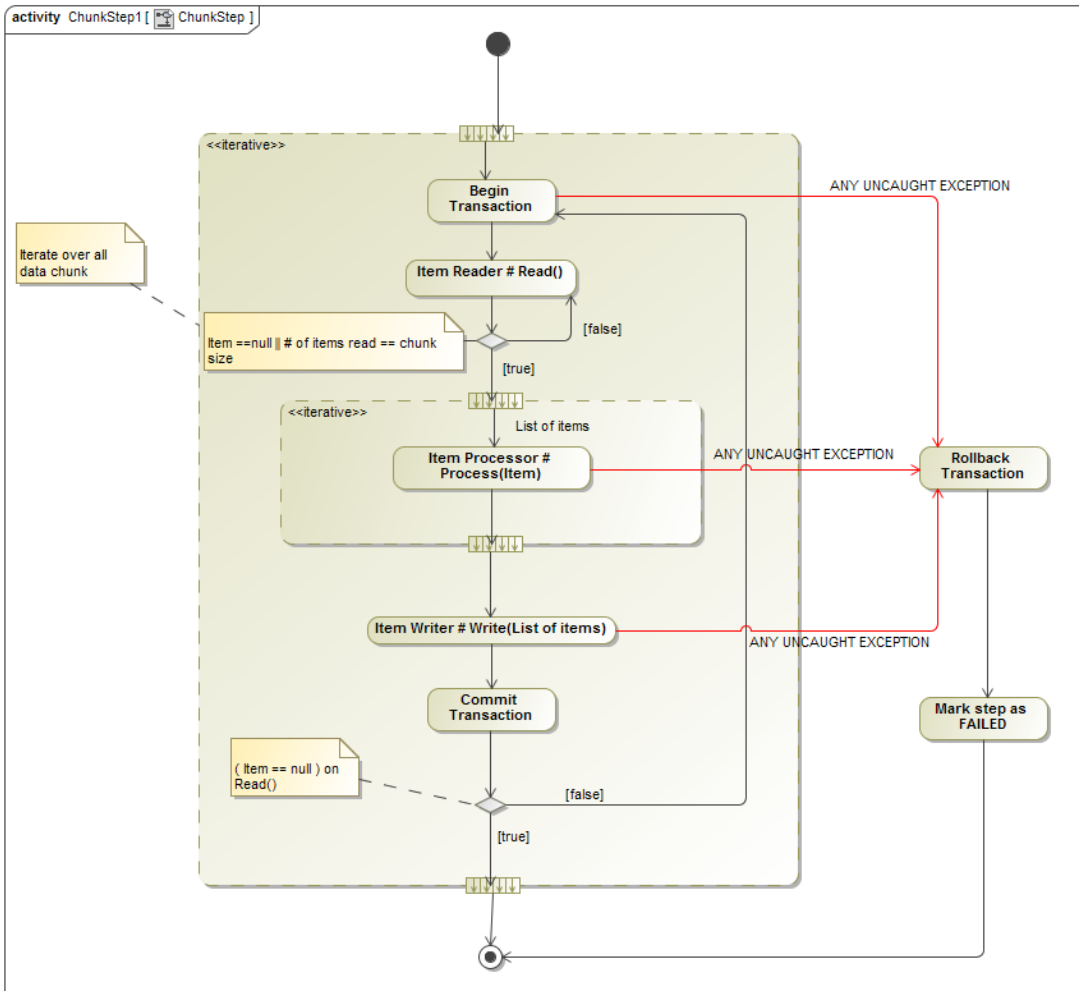

This is the basic behaviour, when everything runs smoothly and the step completes gracefully. The Read/Process/Write pattern is running within the transaction boundaries.

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

Now, what if something goes wrong ?
ANY uncaught exception that is thrown during either Read or Process or Write operation will lead to rollback the current chunk transaction; this will cause the step to FAIL, and consequently, the job to FAIL. If the job restartability has been set up, the job repository will be updated on the transaction rollback, so that the job can be restarted at a later time.

The figure below illustrates that mechanism

**Figure 3.2. Chunk oriented step : when the transaction rollback occurs**




## Caution

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

# The tasklet approach

Using the 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 the `Summer.Batch.Core.Step.Tasklet.ITasklet` interface.

The transactional support is guaranted by the `Summer.Batch.Core.Step.Tasklet.TaskletStep` class, that is in charge of executing the batchlet code (see the `DoExecute` method, that delegates to the `DoInTransaction` method, that wraps the tasklet code effective execution (the `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 the examples of this section.

## Job Configuration

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

There are three kinds of batch elements: steps, flows, and splits. Steps are the 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 the job repository is persisted (see section Job Explorer and Job Repository), the job restartability can be set with the `restartable` attribute. The 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 the `listeners` element. The listeners must implement the `Summer.Batch.Core.IJobExecutionListener` interface and be registered with the corresponding id in the Unity container.

### Example 4.3. Job Configuration

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

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

# Step Element

A step is defined using the `step` element and the `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. The step is divided in chunks: records are read until the chunk size has been reached, then they are processed, and the processed items are finally written. Chunks are repeated until there are no more records to read. Each chunk is executed in a transaction.

The `chunk` element has a required attribute, `item-count`—which specifies the chunk size—, and must contain the `reader` and `writer` elements. The `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 the `IItemReader` interface, the processor must implement the `IItemProcessor` interface, and the writer must implement the `IItemWriter` interface. All must be registered with the corresponding id in the Unity container.

## Batchlet Steps

A batchlet is a step that is entirely delegated to a C# class, that must implement the `ITasklet` interface. The `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 the `listeners` element. The listeners must implement the `IStepExecutionListener` interface and be registered with the corresponding id in the 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>
```

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

## Partitioning

Partitioning allows the execution of several instances of a step using a task executor. A partitioner, implementing the `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 the Unity id of the partitioner with the `ref` attribute and the grid size (which will be provided to the partitioner as a parameter) with the `grid-size` attribute. If not specified, the default grid size is 6.

### Note

The way the different step instances are executed depends entirely on the 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 the 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 the different batch elements are executed depends entirely on the 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. The first element is executed, then the 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 the `next` attribute, which contains the id of the 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. The other ones may appear in any order, as only the `next` attribute is relevant. However, it is strongly advised to write them in the correct order.

## Conditional Control Flow

The batch element to execute can also depend on the status of the execution of the previous batch element by using the `next` element instead of the `next` attribute. The `next` element has two required attributes: `on` and `to`. The `on` attributes contains a string representing a status and the `to` contains the id of a batch element. The `next` elements are processed in order: the first that matches the status of the previous execution will designate the next element. If no element matches, the 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 the standard Summer Batch code for step failure. The `*` value in the second `next` element is the usual wildcard for *any value*. The status of a step can be customized using the `AfterStep` method of the step listener.

## End and Fail Control flow

In addition to the `next` element, `end` or `fail` elements can be used. Both terminate the job, but the 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 the `ref` attribute) must be registered in a Unity container using the `UnityLoader` class. It has two protected methods that can be overriden: `LoadConfiguration(IUnityContainer)`, which makes the registrations for the 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 the required registrations by default, thus it is not required to override it unless this default configuration is not appropriate for the current batch.

The only mandatory interface to register is `IJobOperator`, but the 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 the default implementation, `SimpleJobOperator`, but users are free to provide their own implementation.

## Job Explorer and Job Repository

The job explorer manages the persistence of the batch entities while the job repository holds all the associated meta-information. Both can either be persisted in a database or stored in memory. The database persistence is required to restart a job but the 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. The creation (and drop) sql scripts are being provided for the three supported rdbms (MS Sql Server, Oracle database and IBM DB2): see the 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 the `PersistenceSupport` property to choose whether to use database or memory explorer and repository. By default the 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 the 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 the jobs that have been registered in the Unity container. By default an instance of `MapJobRegistry` is used.

## Task Executor

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

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

# Batch Artifacts

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

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

Processors — Processors must implement the `IItemProcessor<TIn, TOut>` interface. The "`TOut Process(TIn)`" method transforms the item returned by the reader (of type `TIn`) to an item of type `TOut` for the writer. If the current item must be skipped, the processor can return `null`. The type of the returned items, `TOut`, must thus be a reference type.

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

Takslets — Takslets must implement the `ITasklet` interface. The "`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 the `IJobExecutionListener` interface. The "`void BeforeJob(JobExecution)`" method is called before the job starts and the "`void AfterJob(JobExecution)`" method is called after the job ends.

Step listeners — Step listeners must implement the `IStepExecutionListener` interface. The "`void BeforeStep(StepExecution)`" method is called before the step starts and the "`ExitStatus AfterStep(StepExecution)`" method is called after the step ends. The `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*: the *singleton* scope and the *step* scope. A scope determines the lifetime of an instance during the execution of the 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 the `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, the `ContainerControlledLifetimeManager` lifetime manager is used.

### Step Scope

Artifacts defined in the step scope use the `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 the step scope (like the reader or the writer) will not be shared by the different executions.

#### Note

When an artifact in the singleton scope references an artifact in the step scope, a proxy is created. When the proxy is used it will retrieve the appropriate instance of the artifact, depending on the current step execution. The 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

the 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 the Unity container implements the `Summer.Batch.Common.Factory.IInitializationPostOperations` interface, the `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 the 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 the job context. The constructor takes the key of the job context value to get as parameter. It the value is not of type `T`, it converts it using its string representation.

`Summer.Batch.Core.Unity.Injection.StepContextValue<T>`

Injects a value from the step context. The constructor takes the key of the step context value to get as parameter. It the value is not of type `T`, it converts it 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 the required type.

The late binding injection parameter value currently supports three sources for the late binding sequence: (1) `jobExecutionContext`, which injects a value from the job context using `JobContextValue`, (2) `stepExecutionContext`, which injects a value from the step context using `StepContextValue`, and (3) `settings`, which injects a string value read from the 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. The 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 the 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 the 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 the configuration of the registration, and the `Register()` method performs the 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 the current registration to allow chained calls. This 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 correcponding parameter value is added to the 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 the following injection parameter values will be used as parameter for the constructor. The actual constructor is inferred from the parameter values at resolve time. There can only be one constructor injection per registration.

`Method(string)` — Specifies method injection. The parameter is the name of the method to call and all the following injection parameter values will be used as parameter for the method. The actual method is inferred from the

| | |
|---|---|
| | name and the parameter values at resolve time. There may be several method injections per registration (even for the same method). |
| `Property(string)` | Specifies property injection. The parameter is the name of the 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 the Unity container *at resolve time*. The injected object is resolved using the type passed as generic parameter and the name passed as parameter. The name is optional. Since the resolution of the 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 the unity container *at resolve time*. The objects are resolved using the types passed as parameter. |
| `References<T>(params string[])` | Injects an array of objects that are resolved from the unity container *at resolve time*. The objects are resolved using the type passed as generic parameter and the names passed as parameter. |
| `References<T>(params Reference[])` | Injects an array of objects that are resolved from the unity container *at resolve time*. The objects are resolved using the `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

## 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 the `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.

The snippet above displays two typical uses of the `JobStarter` class, with the `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, the 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 the eligible executions. In order to get a fine grained control over job executions and be able to restart, stop or abandon a specific one, the 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 the `Summer.Batch.Core.Launch.IJobOperator` interface) using the GetJobOperator(UnityLoader loader, XmlJob job) method.

The `IJobOperator` holds all needed operations to get a full control over the job executions. In particular, the `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 the job operator is not needed.

# Chapter 6. Using Basic Features

### Preliminary note

For the sake of 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 in

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

A line mapper has to implement the

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

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

A default implementation is provided :

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

The mapping is done in a two phases process:

- The read line from the flat file is split into fields, using a tokenizer that must be specified at initialization time (a class that implements the `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-lenght 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 that first phase is a field set.

### Note

see the

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

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

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

`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 the flat file) should have an available mapper.

Now let's see a sample. First, the -- trivial -- 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 the 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 the sample flat file data 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 the data and the target business object is achieved using the following `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 the DateTime handling. The `Summer.Batch.Extra.DateParser` is provided, but you can provide your own implementation to cover more specific needs. Please review the corresponding API doc to see what services are provided by the `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 the Step Scope (`container.StepScopeRegistration`);
- In addition to the mandatory `LineMapper` property, the `FlatFileItemReader` uses :
  - A resource to be read (= the flat file); this is mandatory; In the sample, the resource path is read from the Settings.config file using a key;
  - An optional encoding to specify the flat file encoding; Use the `Encoding`.`GetEncoding` static methods family to provide a proper encoding;
  - Other optional properties not shown in the sample :
    - `LinesToSkip` : if set, the given number of lines will be skipped at the start of the resource;
    - `Strict` : flag for the strict mode; in strict mode, an exception will be thrown if the resource to be read does not exist (vs. a simple warn logged is non-strict mode);

# Using a flat file writer

Provided implementation is the

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

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

    `Summer.Batch.Infrastructure.Item.File.Transform.ILineAggregator<in T>` interface ; this class is responsible of aggregating the business object properties into a single string that can be used to write a line into the target flat file.
- Optional properties (some having default values):
  - `LineSeparator` : the line separator for the lines to write in the flat file;defaults to `System.Environment.NewLine`;
  - `Transactional` : a flag to tell whether 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 tell whether the writer buffer should be flushed after each write; defaults to false;
  - `SaveState`: a flag to tell if the state of the item writer should be saved in the execution context when the Update method is called; defaults to true;
  - `AppendAllowed`: a flag to tell if an existing resource should be written to in append mode; defaults to false;
  - `DeleteIfExists`: a flag to tell if an existing resource should be deleted; if AppendAllowed is set to true, this flag is IGNORED; defaults to false;
  - `DeleteIfEmpty`: a flag to tell 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 the file, if this makes sense; No default value;
  - `FooterWriter`: a footer writer (class implementing the

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

The writing process takes a business object as input, transforms it into a string using the `LineAggregator` and append the string to the 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 the `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 the provided format.);
- `Summer.Batch.Infrastructure.Item.File.Transform.PassThroughLineAggregator<T>`: transforms an object to a string by simply calling the `ToString` method of the object;

Our example uses the `FormatterLineAggregator`, providing a format string through the unity configuration; To fully understand the unity configuration that follows, the 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; }

    }
}
```

Eventually, here comes the 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

- The `FormatterLineAggregator` requires a `Summer.Batch.Infrastructure.Item.File.Transform.IFieldExtractor<in T>` implementation to be provided 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 the property names (using the `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 the target flat file; The order is significant. The properties values will be passed in that order to the `string.Format` method used by the `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 the flat file.

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

# Reading from and writing to RDBMS

## Reading from a database

Support for reading from a database is brought by the

`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.

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

- ConnectionString : a `System.Configuration.ConnectionStringSettings` instance; the connection string is used to provide all required details needed to connect to a given database; Those 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 the 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 the resultset returned by the query execution to a business object of type T.

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

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

Supported syntax for query parameters : the query parameters are identified either with the ':' or the '@' 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 the query are stored in a dictionary ; matching with the query parameters will be done using the dictionary entries keys;
- `Summer.Batch.Data.Parameter.PropertyParameterSource`: The `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 the example is

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

Records are read from the 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
  )
;
```

The results from the 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 the resultset rows and the 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 the 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>
```

Then, the unity configuration part:

### 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 the `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); the SqlQueries class is the corresponding designer artifact created by Visual Studio to wrap the underlying resource.

# Writing to a database

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

- ConnectionString : a `System.Configuration.ConnectionStringSettings` instance; the connection string is used to provide all required details needed to connect to a given database; Those details are usually being stored in an application xml configuration file;
- Query : a string representing the sql query to be executed against the database; To write to a database, only INSERT or UPDATE sql statements should be used;Externalizing the sql queries in a resource file (.resx) is recommended; As for the reader, the query parameters will be prefixed either by ':' or '@'.
- DbParameterSourceProvider : a class that implements the `Summer.Batch.Data.Parameter.IQueryParameterSourceProvider<in T>` interface, in charge of filling the query parameters with the appropriate values, usually from a business object.

  The method to be implemented is

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

Summer Batch provides an implementation for the `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 the query parameters to the business object properties, on a naming convention basis.

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

- AssertUpdates : a flag to indicate whether to check if database rows have actually been updated; defaults to true (which is the 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 the business object and the query parameters is achieved by using a `PropertyParameterSourceProvider`.

Now, we need to configure the writer; first, in the 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>
```

Then the unity configuration snippet:

### 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:

- The connection details are read from an application xml config file;
- The query is read from a resource file (.resx);

# Database Support

Summer Batch currently supports three RDBMS, with the following invariant provider names:

| 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` |

The `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` that is used to replace parameter in queries with the correct placeholder. All SQL queries should use either "@" or ":" to prefix parameters, the placeholder getter will be used to transform the query so that it uses the placeholder required by the provider.

`Incrementer` — An instance of `IDataFieldMaxValueIncrementer` that is used to retrieve unique ids for different batch entities, such as job or step executions. The best way to generate unique ids is very dependent on the actual RDBMS, so the extension is required to provide an incrementer.

The following example show how support for PostgreSQL can be added, using the 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 provide a complete support for an additional RDBMS, the job repository ddl scripts must be adapted to the target RDBMS;

Typical required ddl adjustements are :

- adapting types names to the 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 example to be exhaustive, we provide in the corresponding appendix section, the ddl scripts for PostgreSQL. Those scripts are provided 'AS IS', just for the sake of completion.

# Chapter 7. Using Advanced Features

## Reading and writing EBCDIC files using Cobol copybooks

Legacy Cobol batches involve frequently dealing with EBCDIC files, using copybooks. Modernizing such batches typically requires to be able to read and write those files, given the copybooks. Two classes have been developped 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 that a xml version of a Cobol copybook is 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 the `Summer.Batch.Extra.Copybook.IEbcdicReaderMapper<T>` interface ;
- a path to the ebcdic file to be read (Resource property).

### Caution

The two elements (copybook xml file and ebcdic reader mapper class) are mandatory. Failing to provide either of them will prevent the EbcdicFileReader from working, and the job will likely fail.

The Cobol copybook is used by the reader to extract ebcdic 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>
```

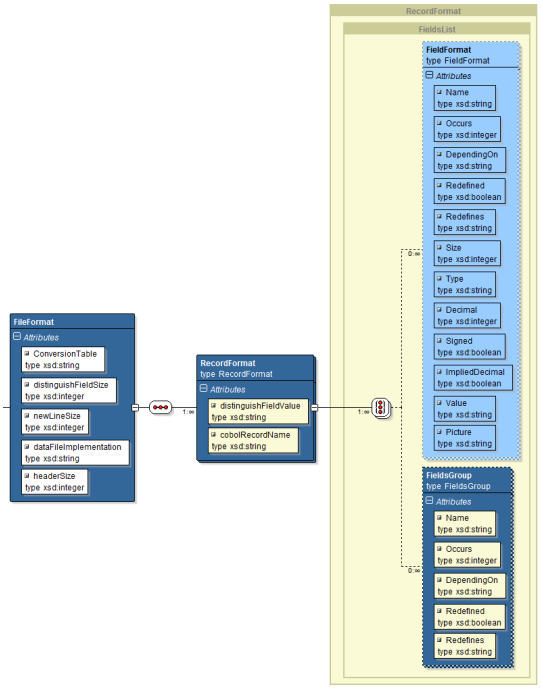
### Note

A remark about the ConversionTable (=encoding) specified in the copybook xml file: this attribute must be given a property that can be understood by the C# API. This requires to use the C# encoding names (which differ from java encoding

names convention -- java uses "Cp037" where C# uses "IBM037" for example). The `System.Text.Encoding` has the `GetEncodings` method to list all available encodings.

The xml copybook file must be compliant with the following 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 the `Summer.Batch.Extra.Copybook` namespace:

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

Then the ebcdic reader mapper is used to transform those records into more convenient business objects. Below is the 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, we need to define a mapper in charge of transforming records into business objects. The mapper must implement the `Summer.Batch.Extra.Ebcdic.IEbcdicReaderMapper<T>` interface. The `Summer.Batch.Extra.Ebcdic.AbstractEbcdicReaderMapper<T>` is a convenient 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;
        }
    }
}
```

Then everything needs to be configured. The EbcdicFileReader is 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

A few details are worth mentioning here:

- The Ebcdic file reader is registered within a step scope, using the `StepScopeRegistration` extension;
- The Ebcdic file reader registration is made using the same name 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 the value read in the 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 must be provided:

- a list of Cobol copybooks xml versions (Copybooks property);
- a class in charge of transforming ebcdic record into a business object (EbcdicWriterMapper property): one can use the `Summer.Batch.Extra.Ebcdic.EbcdicWriterMapper` class or a custom sub-class that inherits from it;
- a path to the ebcdic file to be written (Resource property).

## Caution

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

The three elements (copybook xml files list, ebcdic writer mapper class and path to resource to write to) are mandatory. Failing to provide either of them will prevent the EbcdicFileWriter from working, and the job will likely fail.

Let's review the corresponding needed configuration. 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>
```

Regarding Unity configuration, here is the corresponding part:

### 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

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

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

  The `.Resources()` call (note the extra "s") is able to load into a list of resource paths a semicolon-separated path string.

- The `Summer.Batch.Extra.Ebcdic.EbcdicWriterMapper` is able to automatically convert a business object into a list of values that can be written in an ebcdic record (see the `Summer.Batch.Extra.Ebcdic.EbcdicWriterMapper#Map` method). The automatic mapping between the business object properties and the copybook records is made on a name convention basis. The property name must be equal to the 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 the 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 a basic ftp put operation support;
- `Summer.Batch.Extra.FtpSupport.FtpGetTasklet`: provides a basic ftp get operation support.

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

# Ftp put operations

Using the `FtpPutTasklet`, you can put a given file on 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 use to connect to the ftp host;
- Password : password for the above ftp user.

The following properties are optional (need to be set at initialization time, but a default value is provided):

- Port : ftp host port (if non standard). Defaults to 21;
- RemoteDirectory : path to ftp remote directory. Defaults to empty string, meaning that the defaut 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 here is a sample Unity configuration:

### 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 use to connect to the ftp host;
- Password : password for the above ftp user;

- FileNamePattern : a filename pattern, similar in form to what DirectoryInfo.GetFiles(string) supports to filter the files to be downloaded from the fpt remote directory;
- LocalDirectory : path to local directory where downloaded files will be stored; The AutoCreateLocalDirectory boolean property controls whether the local directory should be automatically created if non-existent (defaults to true); If AutoCreateLocalDirectory is set to false, and the 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 (need to be set at initialization time, but a default value is provided):

- Port : ftp host port (if non standard). 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. The FileNamePattern property will be used to filter the files that need deletion.

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 the `EmailTasklet` (full name : `Summer.Batch.Extra.EmailSupport.EmailTasklet`), you can send email using the SMTP protocol. This tasklet relies on the `System.Net.Mail.SmtpClient` to perform the mail sending 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 : the from email address for sending mails;
- Subject : the subject of the mail to be sent;
- Body : resource path to the mail body content;
- At least a recipient must be specified: the recipients are stored in three separate lists:
  - To : list of direct recipients addresses for the mail;
  - Cc : list of copy recipients addresses for the mail;
  - Bcc : list of hidden copy recipients addresses for the mail;

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

Optional properties (need to be set at initialization time, but a default value is provided if needed):

- Username : if the smtp server requires a user name, this property should contain it; No default value;
- Password : if the smtp server requires a password for the username, this property should contain it; No defaut 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 : a positive or equal to zero 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 here 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

The `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.

A single property is mandatory : the FileToCheck, that 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

A typical usage would make 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

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

Two properties are needed: 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.

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

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

**Reading and writing records.** A record is a unit of information that can be sorted and the sort 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 the size. For separated records, the separator must be set with the `Separator` property. For fixed-length records, the length must be set with the `RecordLength` property. If none of these properties have been set, the tasklet will assume that records have an RDW.

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

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

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

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

**Managing duplicates.** By default, all records that are identical in the context of the sort card will appear in the output. The order among these identical elements in unspecified. If the `SkipDuplicates` property is set to `true`, only one of the identical records will be kept (the selection is also unspecified). It is also possible to sum the lines by setting the `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 the DFSORT syntax, but only the following format 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 the DFSORT syntax and supports the following features:

| | |
|---|---|
| Comparable elements | • Fields<br>• Decimal constants<br>• Character string constants |
| Field formats | • String (CH) and sub-string (SS)<br>• Zoned (ZD)<br>• Packed (PD)<br>• 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 the standard DFSORT "`BUILD`" syntax with the 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 the emulation of main frames' generation data groups (GDG) using a specific resource loader, `Summer.Batch.Extra.IO.GdgResourceLoader`. The 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, the Unity loader must register the 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 the group are identified by a reference number: the file 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 the "`gdg://`" protocol as follows:

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

Where "`<path>`" is the path to the file without the extension or the generation number, "`<number>`" is the reference number of the file, and "`<extension>`" is the extension of the file (which is optional). The final file name will be computed by concatenating the path with the generation number and the extension if there is one. For instance, if the 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 the 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"
```

The "limit" option sets the number of file that should be kept in the group and the "mode" options specify how files are managed once the limit has been reached: in the "empty" all files existing before the current execution are deleted while in the "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 the 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 Summer Batch: ContextManager, ContextManagerUnityLoader, AbstractExecutionListener and AbstractService.

## ContextManager class

The contexts are basically dictionaries. The ContextManager simply enables to store and retrieve from the dictionary and gives some utility methods, particularly methods related to counter management, which is a very common need. There is a single ContextManager class

for both job context and step context but two instances will be available, one for the job and the other for the step.
It implements the `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 `object` for internal compatibility, the key should be a `string` and will be converted to a `string` if it is not.

## `ContextManagerUnityLoader`

In order to use the context managers, one should extend a dedicated UnityLoader: `ContextManagerUnityLoader`, that will register in the unity container two instances of the `ContextManager`, one for the job and one for the step. The keys used for these regsitrations are `BatchConstants.JobContextManagerName` and `BatchConstants.StepContextManagerName`. As usual, this class must be extended and the `LoadArtifacts` must be implemented with all the 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` presented just below.

## `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 the `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 the processor code. Even other services involved in the 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`, the context managers will not be linked to the 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, for example, to be able to read several files at once: the first file will be read

through the reader as usual while the second file will be read during the processing. There are two such classes. The `ProcessReaderAdapter` will supply the `ReadInProcess` method, while the `ProcessWriterAdapter` will supply the `WriteInProcess` method. Both classes have an `Adaptee` property that references the actual reader or writer, and that will be wired through Unity setup, and also 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, the Flat File Writer was presented. Among other properties to set, there was the LineAggregator. The `Summer.Batch.Extra.Template.AbstractTemplateLineAggregator<T>` is dedicated to format the output thanks to an external format file. The class must be extended by implementing the `IEnumerable<object> GetParameters(T obj)` that converts the business object used in the batch into an enumeration of values.

The template file must have a content like this: `Key:Format`, with the Format in a C# fashion. Multiple lines of this form can appear with multiple keys, and a format can be multi line by repeating only the 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 the resource file containing the format to use;
- TemplateId : The key to the correct format in the file.

Optional properties (need to be set at initialization time, but a fallback behavior is available):

- InputEncoding : the template file encoding, defaults to `Encoding.Default`;

- Culture : the template file culture, default to `CultureInfo.CurrentCulture`;
- LineSeparator : the line separator to use 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 mostly 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 explined 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 the same template file. This is not very useful. 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 of all, the writer and with the 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 the 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);
    …
}
…
```

Of course, the `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

## S



## T



## U



## X