SummerBatch/Getting started

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

Table of Contents

Pre-Requisites

- Visual Studio 2013 version or better (target framework is .NET 4.5);
- We recommend being familiar with C# development using Visual Studio;
- An internet connection (some resources are hosted on separate sites).

Project Setup

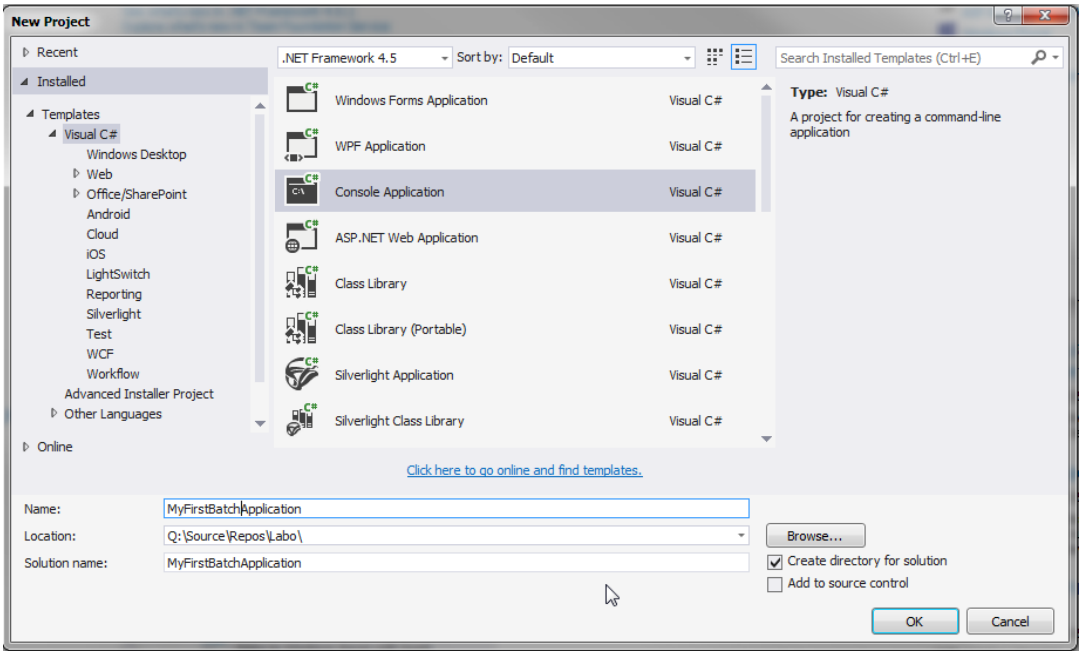
Note

For the impatients:

- the complete ready-to-use solution is available and browsable from the corresponding github Summer Batch samples repository.

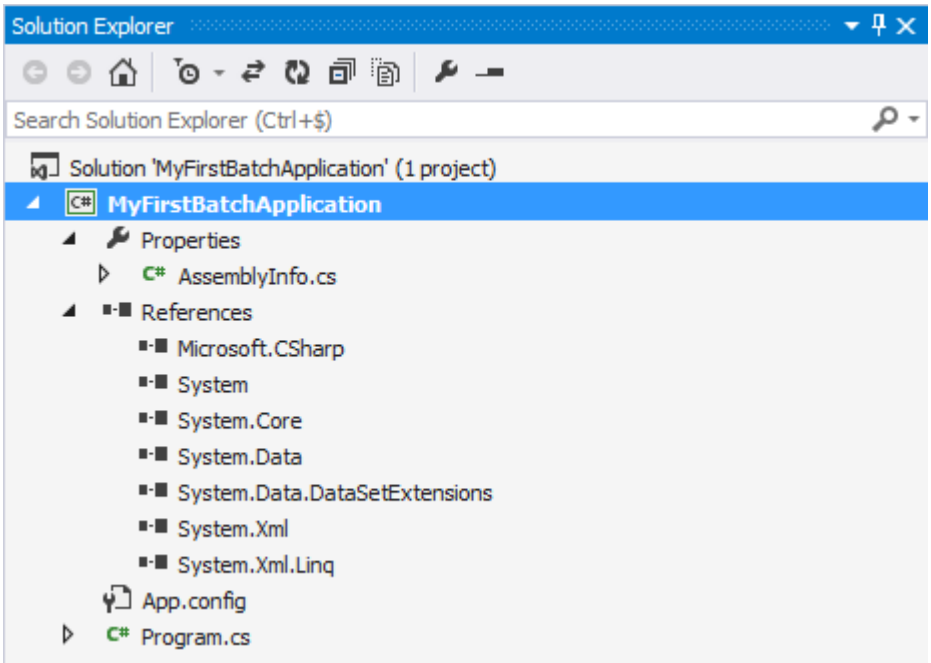
Let's get started. In VS, create a new 'Console Application' project (found in the Visual C# Templates), giving a meaningful name; we chose to name it 'MyFirstBatchApplication';

Figure 1. Creating a Console Application in VS



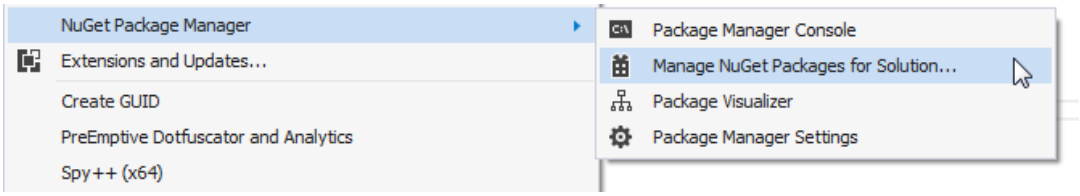
In the Solution explorer, you should see a solution (that has been automatically created) containing the freshly created project:

Figure 2. Freshly created project



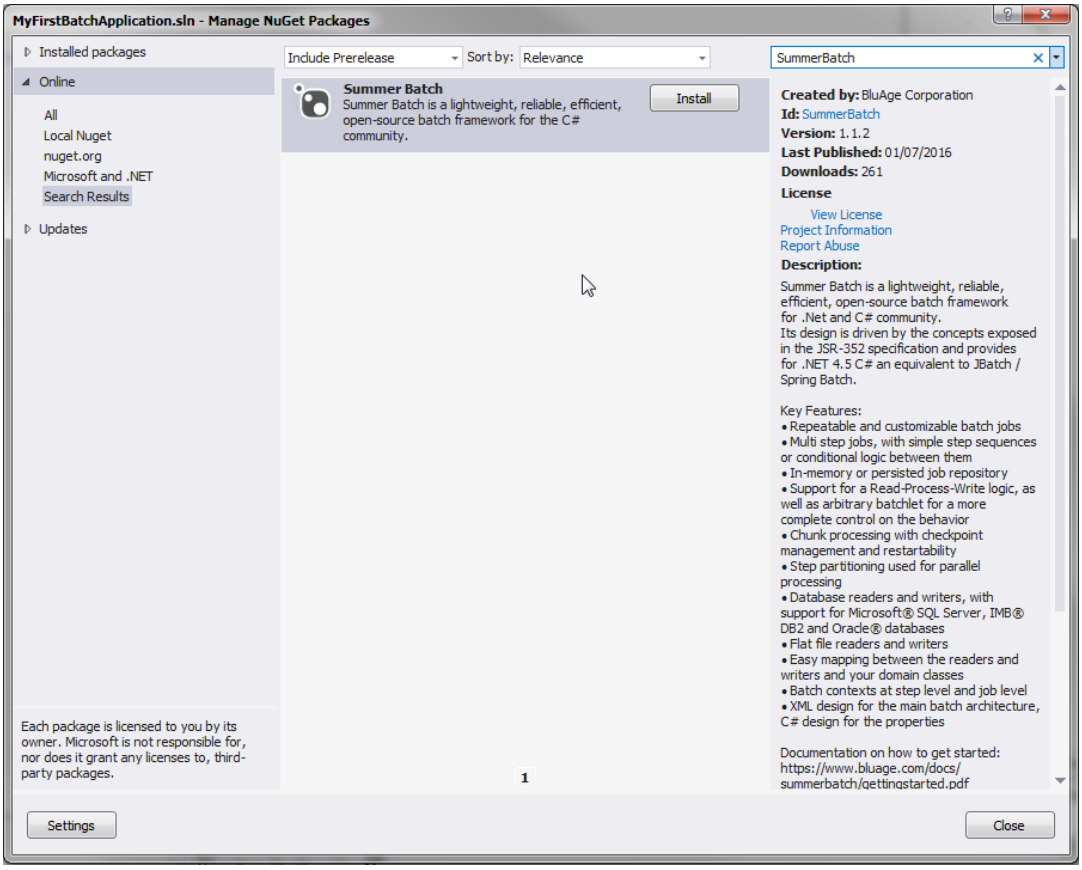
In the freshly created project, add the dependency to Summer Batch using the nuget package manager :

Figure 3. Launch the NuGet Package Manager



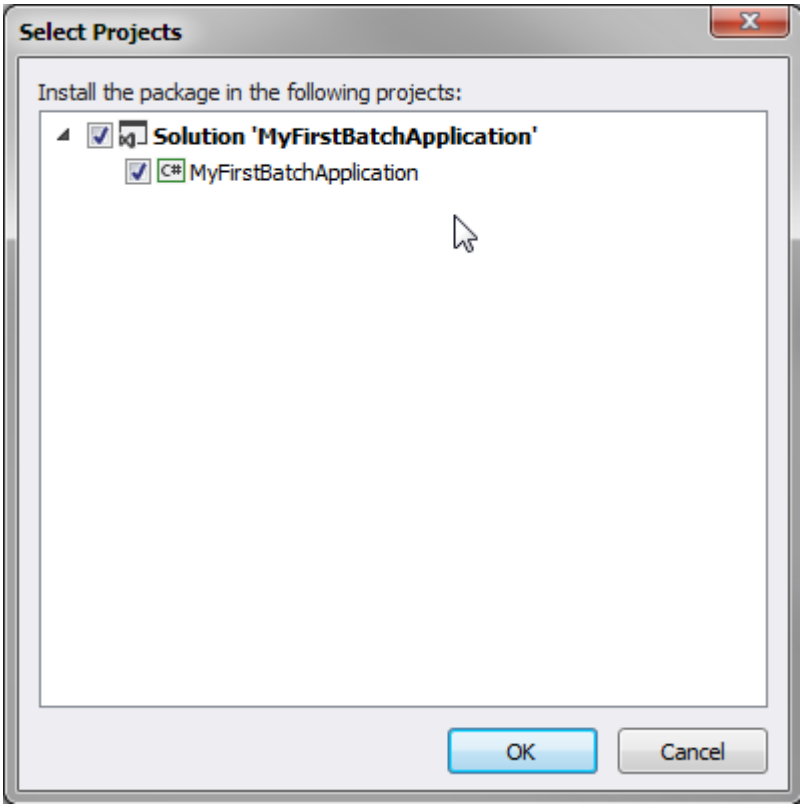
There, look for the SummerBatch package (use the search textbox), then click on the Install button.

Figure 4. Install the Summer Batch Nuget package - part 1



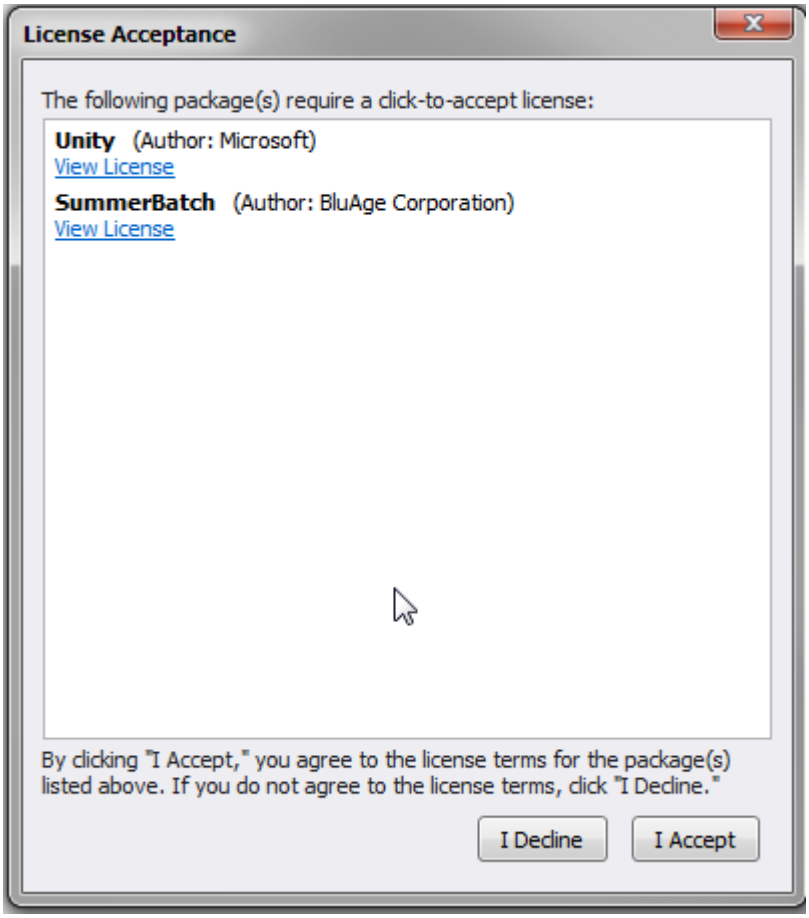
On the next dialog, leave your project and solution ticked, then click on the OK button

Figure 5. Install the Summer Batch Nuget package - part 2



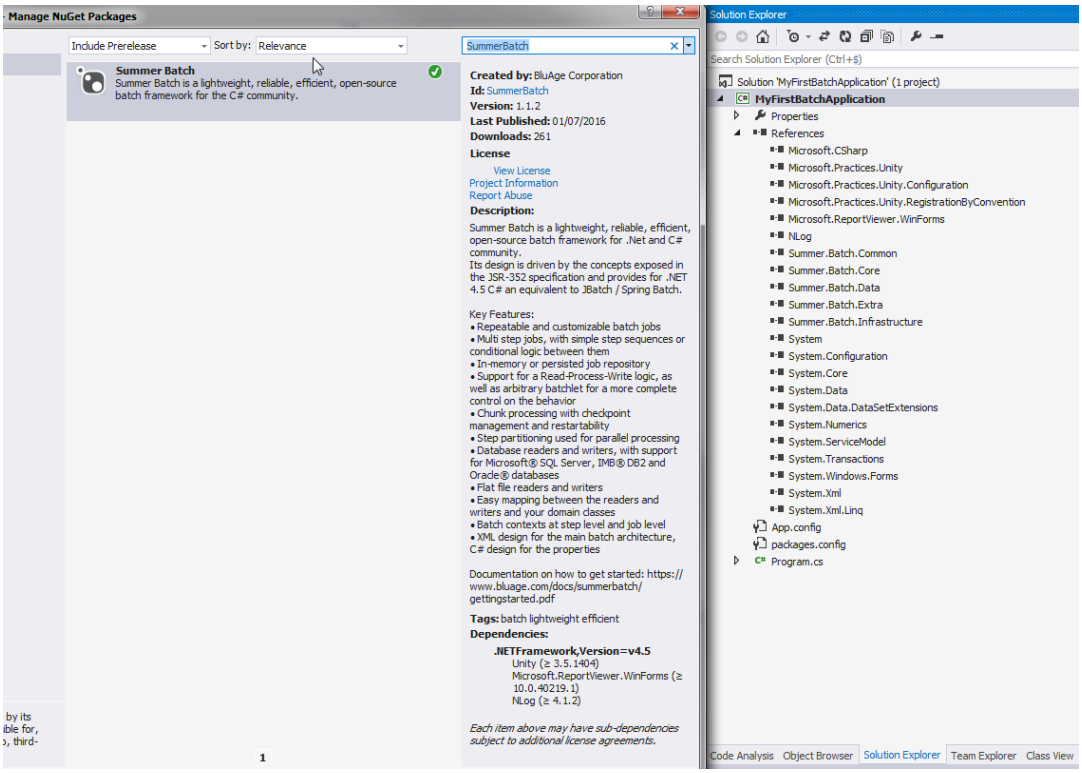
On the next dialog, you will be prompted to accept licenses bound to the components being installed (Unity and SummerBatch). Click on the 'I Accept' button :

Figure 6. Install the Summer Batch Nuget package - part 3s



Once installation has been successfully performed, you'll end with a this Dialog. Just dismiss it, using the Close Button. Note all the additional references in the solution explorer, brought by the SummerBatch nuget package.

Figure 7. Install the Summer Batch Nuget package - part 4



Project setup is done. Now it's time to code !

Business Code

The goal of this batch is to persist an input flat file (semicolon separated fields records on each line) to a rdbms table. The records in the flat file are using the following structure:

```
code(integer);name(string);description(string);
  date(a string representing a date, using the yyyyMMdd format)
```

Here are some sample data, to illustrate the structure :

```
1;FlatFile1 ; FlatFileDesc1 ;19700731
2;FlatFile2 ; FlatFileDesc2 ;20150101
```

This is a typical repetitive Read/Process/Write pattern scenario, a very common batch use case;

The plan is to use a chunk oriented step with the following elements :

- A flat file reader, that will read records from the flat file and store them into dedicated business objects: READING;
- A processor, that will transform the business records according to some business logic: PROCESSING;
- A database writer, that will dump the processed business objects into a rdbms table: WRITING.

Let's begin with the reading part.

Reading the flat file

We'll be using a `Summer.Batch.Infrastructure.Item.File.FlatFileItemReader<T>` for this purpose;

The `FlatFileItemReader` is a versatile class that requires some business code elements :

1) A business object to store read records from the flat file : `FlatFileRecord.cs`; Create a new folder named for example 'Business' in your project, to hold this business object class;

Here is the source code for this class :

```
using System;

namespace MyFirstBatchApplication.Business
{
    /// <summary>
    /// FlatFileRecord : business object to store the
    /// read records from the flat file
    /// </summary>
    [Serializable]
    public class FlatFileRecord
    {
        /// <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; }

    }
}
```

Note

This is a straightforward example, where each flat file record field is intended to be mapped to a business object property. Therefore, the business object mimics the flat file structure (property names and types).

2) A field set mapper : the field set mapper is in charge of mapping the record fields to the business object properties. By contract, it must implement the `Summer.Batch.Infrastructure.Item.File.Mapping.IFieldSetMapper<T>` interface. The method to be implemented (`MapFieldSet`) is supposed to be reading the record fields (using their position) to fill the corresponding business object properties, making necessary transformations if required.

Let's create a sub-folder named 'Mappers' in the 'Business' folder, to store the `FlatFileRecordMapper` class, whose code follows :

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

namespace MyFirstBatchApplication.Business.Mappers
{
    public class FlatFileRecordMapper : IFieldSetMapper<FlatFileRecord>
    {
        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="FlatFileRecord" />.
        /// <param name="fieldSet">the field set to map</param>
        /// <returns>the corresponding item</returns>
        /// </summary>
        public FlatFileRecord MapFieldSet(IFieldSet fieldSet)
        {
            // Create a new instance of the current mapped object
            return new FlatFileRecord
            {
                Code = fieldSet.ReadInt(0),
                // ReadString trims the read string, use ReadRawString
                // to keep trailing spaces
                Name = fieldSet.ReadString(1),
                Description = fieldSet.ReadString(2),
                Date = _dateParser.Decode(fieldSet.ReadString(3))
            };
        }
    }
}
```

Note

`Date = _dateParser.Decode(fieldSet.ReadString(3))` : the date is read as a string, so a conversion is needed to have the corresponding DateTime? value; the `Summer.Batch.Extra.DateParser` provides this conversion service (is uses the "yyyyMMdd" format).

We're done with the reading business code crafting. The rest of the mandatory elements required by the `FlatFileItemReader` are standard elements, and only need some Unity configuration, that will be made later on.

Processing the read objects

Usually, the data needs to be transformed, according to some business logic, after having been read and before being written. This transformation is usually done by the processor. For our example, the processor will be implementing the following logic : compare Name and Description of the input business object. If Name equals Description, the replace Description value by "Missing Description"; then return the modified business object.

The processor has to implement the `Summer.Batch.Infrastructure.Item.IItemProcessor<in TIn, out TOut>` interface. Let's create a 'Service' folder, to hold the `FlatFileRecordProcessor` class, whose code is given below :

```
using MyFirstBatchApplication.Business;
using NLog;
using Summer.Batch.Infrastructure.Item;

namespace MyFirstBatchApplication.Service
{
    /// <summary>
    /// Implements <see cref="IItemProcessor{TIn, TOut}" />
    /// for FlatFileRecord processing duty.
    /// </summary>
    public class FlatFileRecordProcessor :
     IItemProcessor<FlatFileRecord, FlatFileRecord>
    {

        private static readonly Logger Logger =
         LogManager.GetCurrentClassLogger();

        private const string MissingDescription =
          "Missing Description";

        /// <summary>
        /// Implements the business logic for
        /// FlatFileRecord processing;
        /// </summary>
        /// <param name="item">the item to process</param>
        /// <returns>the item that might have been modified
        /// by the processing</returns>
        public FlatFileRecord Process(FlatFileRecord item)
        {
            Logger.Debug("Treating item with code {0}",
                item !=null && item.Code !=null ?
                item.Code.ToString():"null item or null code item");
            if (item != null && item.Name != null
             && item.Description != null
             && item.Name.Equals(item.Description))
            {
                Logger.Debug("Missing description for item {0} ",
                 item.Code != null ?item.Code.ToString()
                  :"null item code");
                item.Description = MissingDescription;
            }
            return item;
        }
    }
}
```

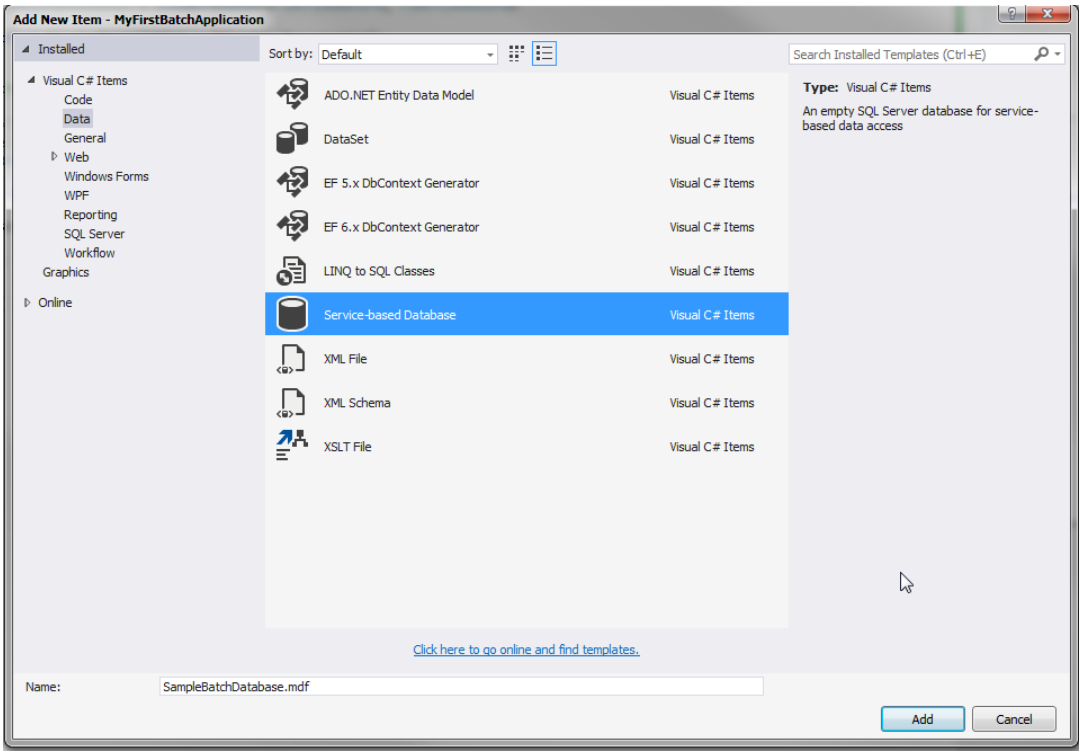
Note

In this example, the input and output of the processor share the same type; this is a common scenario, but obviously not mandatory.

Writing the list of processed objects

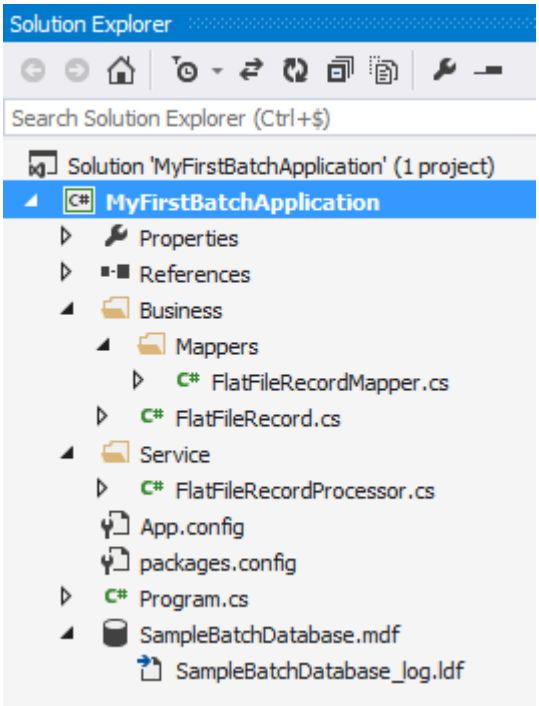
Once the objects have been read and processed, we want to store them in a rdbms table. For our example, we'll use a simple Service-based Database (SQL Server), named 'SampleBatchDatabase.mdf';

Figure 8. Create a new Service-based Database



At this stage, the solution content should look like this :

Figure 9. Create a new Service-based Database

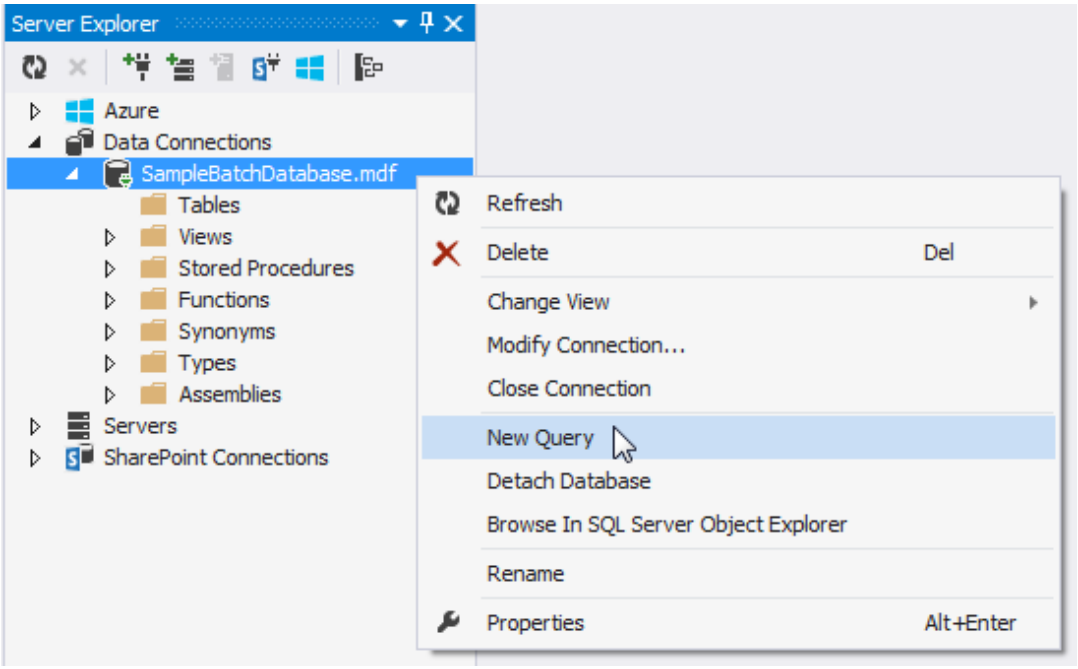


The DDL script to create the table is given below :

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

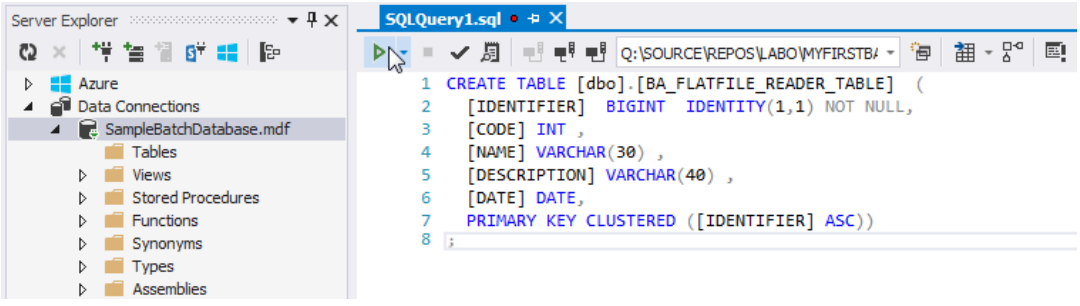
To run the DDL query, you can use the 'New Query' menu entry (from the Server Explorer tab)

Figure 10. New query



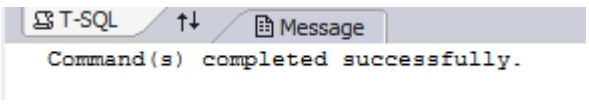
In the opened query window, paste the DDL content, then click on the 'Execute' (green arrow) button

Figure 11. Run the DDL Query



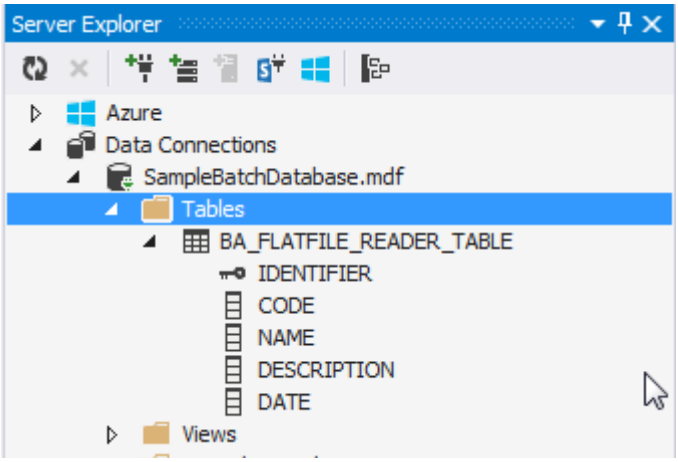
You should see a `'Command(s) completed successfully'` in the Message window

Figure 12. Command completed



Checking the Server Explorer window (might need to be refreshed), you'll see that the table has been created, with the proper columns.

Figure 13. Table has been created



Now that the target rdbms has been set up, we can write in it. We'll be using a `Summer.Batch.Infrastructure.Item.Database.DatabaseBatchItemWriter<in T>` for that purpose. Since we'll only be using standard Summer Batch classes, no additional business coding is required, only some Unity configuration that will done later on.

To be able to write in the table, an update query is needed (INSERT or UPDATE sql statement); We'll be using the following query :

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

Note

In the query, the ':' prefix is used to identify the query parameters. The query parameters names are significant, since they will be automatically mapped to the input business object properties, on a name convention basis.

At this stage, all the business coding has been achieved. It is time to configure the batch, to be able to run it.

Configure and Run

The last steps concern the batch configuration, that involves two elements :

- The job xml configuration;
- The Unity configuration;

Job xml configuration

We believe the configuration should be in a folder of its own; let's create a folder named 'Batch'; The job xml configuration is quite straightforward : the job is made of a single chunk oriented step, holding the reader / processor / writer declaration. The xml file, named 'MyFirstBatch.xml' content is given below :

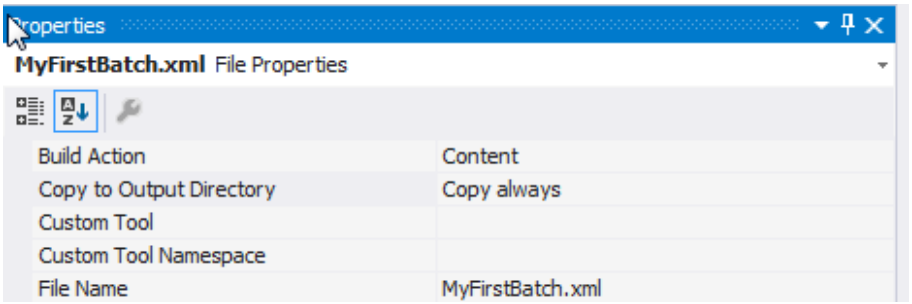
```
<?xml version="1.0" encoding="UTF-8"?>
<job id="FLATFILE_READER_DB_WRITER"
 xmlns="http://www.summerbatch.com/xmlns"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.summerbatch.com/xmlns http://www.summerbatch.com/xmlns/SummerBatchXML_1_0.xsd">
 <step id="FlatFileReader">
  <chunk item-count="1000">
   <reader ref="FlatFileReader/FlatFileReader" />
   <processor ref="FlatFileReader/Processor" />
```

```
    <writer ref="FlatFileReader/DatabaseWriter" />
  </chunk>
 </step>
</job>
```

Note

- Please be sure to select the 'Copy Always' menu entry of the 'Copy to Output Directory' options, in the Properties view for the job xml configuration file (default is 'Do not copy' which will cause runtime issues)

Figure 14. Setting properties for the job xml configuration file



- The `item-count="1000"` attribute value indicates that the chunk size is set to 1000 ; transactions will be commited every time 1000 records have been written or that there is no more data to be treated.

Database access configuration

In order for the code to be able to access the database, some configuration has to be done. This can be achieved in several ways, but the most straightforward is to use the App.config facility; The App.config xml file has been created at project creation time; Let's edit this file to add a `connectionStrings` configuration entry:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <startup>
        <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5" />
    </startup>
  <connectionStrings>
    <add name="Default"
     providerName="System.Data.SqlClient"
     connectionString="Data Source=(LocalDB)\v11.0;AttachDbFilename=|DataDirectory|SampleBatchDatabase.mdf;Integrated Security=True" />
  </connectionStrings>
</configuration>
```

Note

- The connection to the database will be known as "Default", and will be used in the Unity configuration class (see below);
- the `|DataDirectory|` is a shortcut that, since no additional related setup has been done, points at the project root folder. Some additional information can be found HERE (see the end of the article).

Unity configuration

Summer Batch relies on Unity regarding the Dependency Injection mechanism. The Unity configuration is the place where all the batch technical setup will be achieved. Let's create a class named `MyFirstBatchUnityLoader` in the 'Batch' folder.

The source code follows:

```
using System.Configuration;
using System.Text;
```

```
using Microsoft.Practices.Unity;
using MyFirstBatchApplication.Business;
using MyFirstBatchApplication.Business.Mappers;
using MyFirstBatchApplication.Service;
using Summer.Batch.Common.IO;
using Summer.Batch.Core.Unity;
using Summer.Batch.Data.Parameter;
using Summer.Batch.Infrastructure.Item;
using Summer.Batch.Infrastructure.Item.Database;
using Summer.Batch.Infrastructure.Item.File;
using Summer.Batch.Infrastructure.Item.File.Mapping;
using Summer.Batch.Infrastructure.Item.File.Transform;

namespace MyFirstBatchApplication.Batch
{
    /// <summary>
    /// Batch unity configuration
    /// </summary>
    public class MyFirstBatchUnityLoader : UnityLoader
    {
        /// <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)
        {
            //Connection string
            var writerConnectionstring = ConfigurationManager.ConnectionStrings["Default"];

            //input file
            var inputFileResource = new FileSystemResource("data/input/FlatFile.txt");

            // Reader - FlatFileReader/FlatFileReader
            container.StepScopeRegistration<IItemReader<FlatFileRecord>,
             FlatFileItemReader<FlatFileRecord>>("FlatFileReader/FlatFileReader")
                .Property("Resource").Value(inputFileResource)
                .Property("Encoding").Value(Encoding.GetEncoding("UTF-8"))
                .Property("LineMapper")
                 .Reference<ILineMapper<FlatFileRecord>>("FlatFileReader/FlatFileReader/LineMapper")
                .Register();

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

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

            // Field set mapper
            container.RegisterStepScope<IFieldSetMapper<FlatFileRecord>,
             FlatFileRecordMapper>("FlatFileReader/FlatFileReader/FieldSetMapper");

            // Processor - FlatFileReader/Processor
            container.RegisterStepScope<IItemProcessor<FlatFileRecord, FlatFileRecord>,
             FlatFileRecordProcessor >("FlatFileReader/Processor");

            // Writer - FlatFileReader/DatabaseWriter
            container.StepScopeRegistration<IItemWriter<FlatFileRecord>,
             DatabaseBatchItemWriter<FlatFileRecord>>("FlatFileReader/DatabaseWriter")
                .Property("ConnectionString").Instance(writerConnectionstring)
                .Property("Query").
                 Value("INSERT INTO BA_FLATFILE_READER_TABLE (CODE,NAME,DESCRIPTION,DATE)"
                 +" VALUES (:code,:name,:description,:date)")
                .Property("DbParameterSourceProvider").
                 Reference<PropertyParameterSourceProvider<FlatFileRecord>>()
                .Register();

        }
    }
}
```

Note

The forthcoming paragraphs review in details the different Unity configuration parts; reading this can be done later if you are eager to see your first Summer Batch up and running.
In such a case, you can jump directly to the Job Launcher section below.

```
//Connection string
var writerConnectionstring = ConfigurationManager.ConnectionStrings["Default"];
```

This refers to the database configuration access configuration, exposed in the previous section. This ConnectionStrings is consumed by any component that need to access the database; in our example, this concerns the writer only.

```
//input file
var inputFileResource = new FileSystemResource("data/input/FlatFile.txt");
```

This points at the resource that is read by the `FlatFileItemReader`; this setting points at a local folder within the project but one could use an external path as well.

```
// Reader - FlatFileReader/FlatFileReader
container.StepScopeRegistration<IItemReader<FlatFileRecord>,
 FlatFileItemReader<FlatFileRecord>>("FlatFileReader/FlatFileReader")
    .Property("Resource").Value(inputFileResource)
    .Property("Encoding").Value(Encoding.GetEncoding("UTF-8"))
    .Property("LineMapper")
     .Reference<ILineMapper<FlatFileRecord>>("FlatFileReader/FlatFileReader/LineMapper")
    .Register();

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

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

// Field set mapper
container.RegisterStepScope<IFieldSetMapper<FlatFileRecord>, FlatFileRecordMapper>
 ("FlatFileReader/FlatFileReader/FieldSetMapper");
```

The reader configuration is the most tricky part, as it requires a lot of inner components;

```
container.StepScopeRegistration<IItemReader<FlatFileRecord>,
FlatFileItemReader<FlatFileRecord>>("FlatFileReader/FlatFileReader")
```

Caution

The registration must be done using a name that is equal to the ref. attribute set in the job xml configuration : `"FlatFileReader/FlatFileReader" # <reader ref="FlatFileReader/FlatFileReader" />`

This is a general rule, that applies to the processor and the writer of the chunk oriented step; The ref. names set in the job xml configuration file must ALWAYS find their counterparts in the Unity configuration class. The opposite is not true; some elements are registered within the Unity configuration class but are not referenced in the job xml configuration (for example : the `ILineTokenizer` or the `IFieldSetMapper` below)

In a chunk oriented step configuration, the components need usually to be declared within a step scope; this is achieved using the Summer Batch Unity extension (`StepScopeRegistration` method -- when properties values must be set -- or `RegisterStepScope` method -- when no property needs to be set --);

The registration is taking two arguments : the first argument is the target interface, the second argument is the concrete implementation : we use the Summer Batch `FlatFileItemReader`.

Some properties of the flat file reader are mandatory:

- `Resource`: the flat file resource to be read (defined above);
- `LineMapper`: the `ILineMapper` is responsible to map the lines read from the flat file to the target business object; We use a `DefaultLineMapper` that is using an `ILineTokenizer` and an `IFieldSetMapper`:

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

  - `ILineTokenizer`: we use the `DelimitedLineTokenizer`, with the semi-colon delimiter

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

  - `IFieldSetMapper`: we use the `FlatFileRecordMapper` that we wrote earlier :

```
// Field set mapper
container.RegisterStepScope<IFieldSetMapper<FlatFileRecord>,
 FlatFileRecordMapper>("FlatFileReader/FlatFileReader/FieldSetMapper");
```

In addition, we set the `"Encoding"` property to precise the encoding that will be used during the read process (here "UTF8");

The comes the processor and writer registrations, which are much simpler (less plumbering required).

Registering the processor -- nothing particular here, just make sure to use the proper types for in and out generic types for the processor --:

```
// Processor - FlatFileReader/Processor
container.RegisterStepScope<IItemProcessor<FlatFileRecord, FlatFileRecord>,
 FlatFileRecordProcessor >("FlatFileReader/Processor");
```

Note

Since no additional property has to be set, registration is done using the direct `RegisterStepScope` method rather than the `StepScopeRegistration<>().[...].Register()` idiom;

Registering the writer :

```
// Writer - FlatFileReader/DatabaseWriter
container.StepScopeRegistration<IItemWriter<FlatFileRecord>,
 DatabaseBatchItemWriter<FlatFileRecord>>("FlatFileReader/DatabaseWriter")
 .Property("ConnectionString").Instance(writerConnectionstring)
 .Property("Query")
  .Value("INSERT INTO BA_FLATFILE_READER_TABLE (CODE,NAME,DESCRIPTION,DATE)"
  +" VALUES (:code,:name,:description,:date)")
 .Property("DbParameterSourceProvider")
  .Reference<PropertyParameterSourceProvider<FlatFileRecord>>()
.Register();
```

Note

All the properties set for the writer are mandatory :

- the `"ConnectionString"` refers to the previously defined connection to the database;

- the `"Query"` is the sql statement used to write to the database; This could easily be stored in a resource file rather than being hardcoded here to enforce a stronger delimitation between functional and technical code.

- the `"DbParameterSourceProvider"`: this element is in charge of filling the query parameters with the ad-hoc values; the `PropertyParameterSourceProvider<FlatFileRecord>` is furnishing this service by binding the `FlatFileRecord` object properties to the query parameters, on a name match convention basis (matching is case insensitive for the `Property-ParameterSourceProvider`, thus the `":code"` query parameter will be matched with the `"Code"` property; Please note that other `IQueryParameterSource-Provider` implementations may have a different behaviour regarding casing).

Now that the whole configuration has been set up, it's time to write the job launcher to be able to run the batch.

The job launcher

When a new console application project is created within Visual Studio, an entry point class is automatically created : `Program.cs`; Let's modify this class to use it to launch the job. The source code follows :

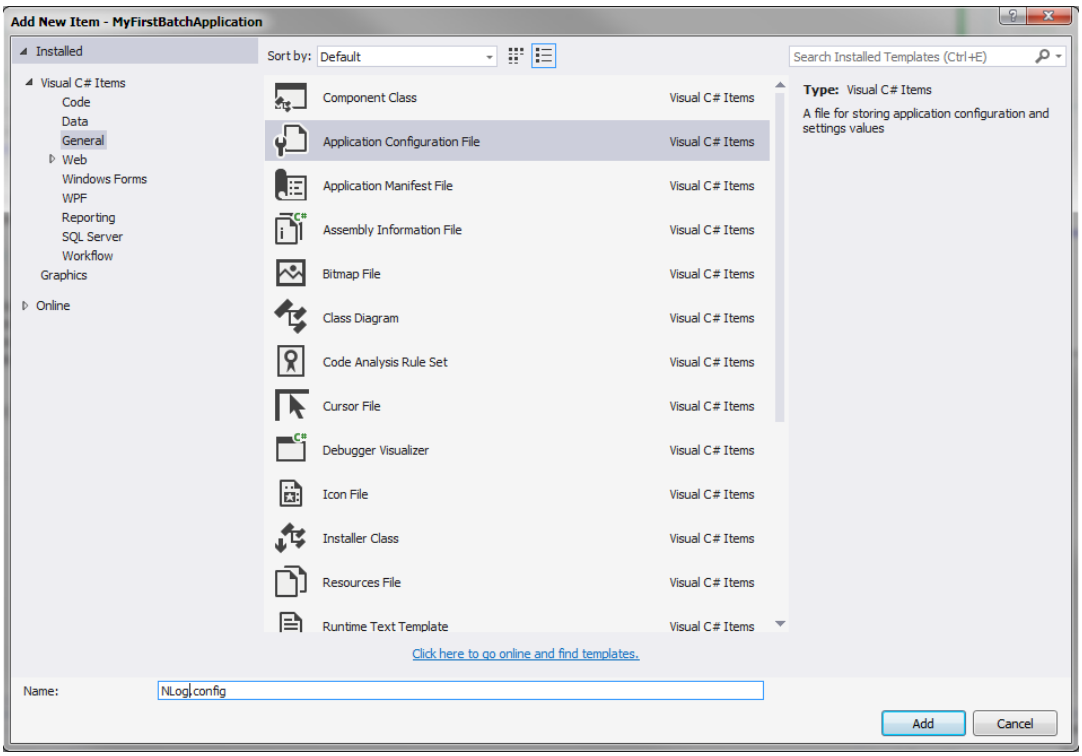
```
using System;
using System.Diagnostics;
using MyFirstBatchApplication.Batch;
using Summer.Batch.Core;

namespace MyFirstBatchApplication
{
    /// <summary>
    /// Main entry point
    /// </summary>
    public class Program
    {
        /// <summary>
        /// Launches the job
        /// </summary>
        /// <param name="args">The arguments for the job execution</param>
        /// <returns></returns>
        public static int  Main(string[] args)
        {
#if DEBUG
            var stopwatch = new Stopwatch();
            stopwatch.Start();
#endif
            JobExecution jobExecution = JobStarter.Start(@"Batch\MyFirstBatch.xml",
             new MyFirstBatchUnityLoader());

#if DEBUG
            stopwatch.Stop();
            Console.WriteLine(Environment.NewLine + "Done in {0}ms.",
             stopwatch.ElapsedMilliseconds);
            Console.WriteLine("Press a key to end.");
            Console.ReadKey();
#endif

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

The entry point uses the `JobStarter` facility to launch the job, using the job xml configuration file and the unity configuration class. In addition, some directives are being used to add some debug facilities (stopwatch and wait for key in the console before closing execution);

Before launching the job, one last configuration step is required : configuring the Logger. Summer Batch relies on NLog, so you must provide a valid NLog configuration file to see the log messages. The configuration file is named 'NLog.config' and is located at the project root.

Details about writing a valid NLog configuration file can be found HERE
Use the 'Add New Item > General > Application Configuration File' to create it :

Figure 15. Setting properties for the job xml configuration file



Here is a sample content :

```
<?xml version="1.0" encoding="utf-8" ?>
<nlog xmlns="http://www.nlog-project.org/schemas/NLog.xsd" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" >

  <targets>
    <target name="console" xsi:type="Console"
     layout="${time} ${level} ${threadid} ${callsite:className=False:fileName=True:methodName=False:includeSourcePath=False}
     ${message} ${exception:format=tostring:maxInnerExceptionLevel=2}" />
    <target name="logfile" xsi:type="File" fileName="C:\temp\MyFirstBatchApplicationLog.txt"
     layout="${time} ${level} ${threadid} ${callsite:className=False:fileName=True:methodName=False:includeSourcePath=False}
     ${message} ${exception:format=tostring:maxInnerExceptionLevel=2}" />
  </targets>

  <rules>
    <logger name="*" minlevel="Debug" writeTo="console" />
    <logger name="*" minlevel="Debug" writeTo="logfile" />
  </rules>
</nlog>
```

Note

- As for the job xml configuration file, make sure to use the 'Copy Always' option to ensure the log config file will be used at run time;
- The minlevel set to 'Debug' is verbose, and is definitely not intended to be used in a production environment. It is aimed at helping developpers to tune their code during the test phase before going to production.

Running the batch

To run the batch, you must provide an input flat file; The flat file is expected to be named 'FlatFile.txt', and should be located in the `data/input` folder.

Here is a sample FlatFile.txt content :

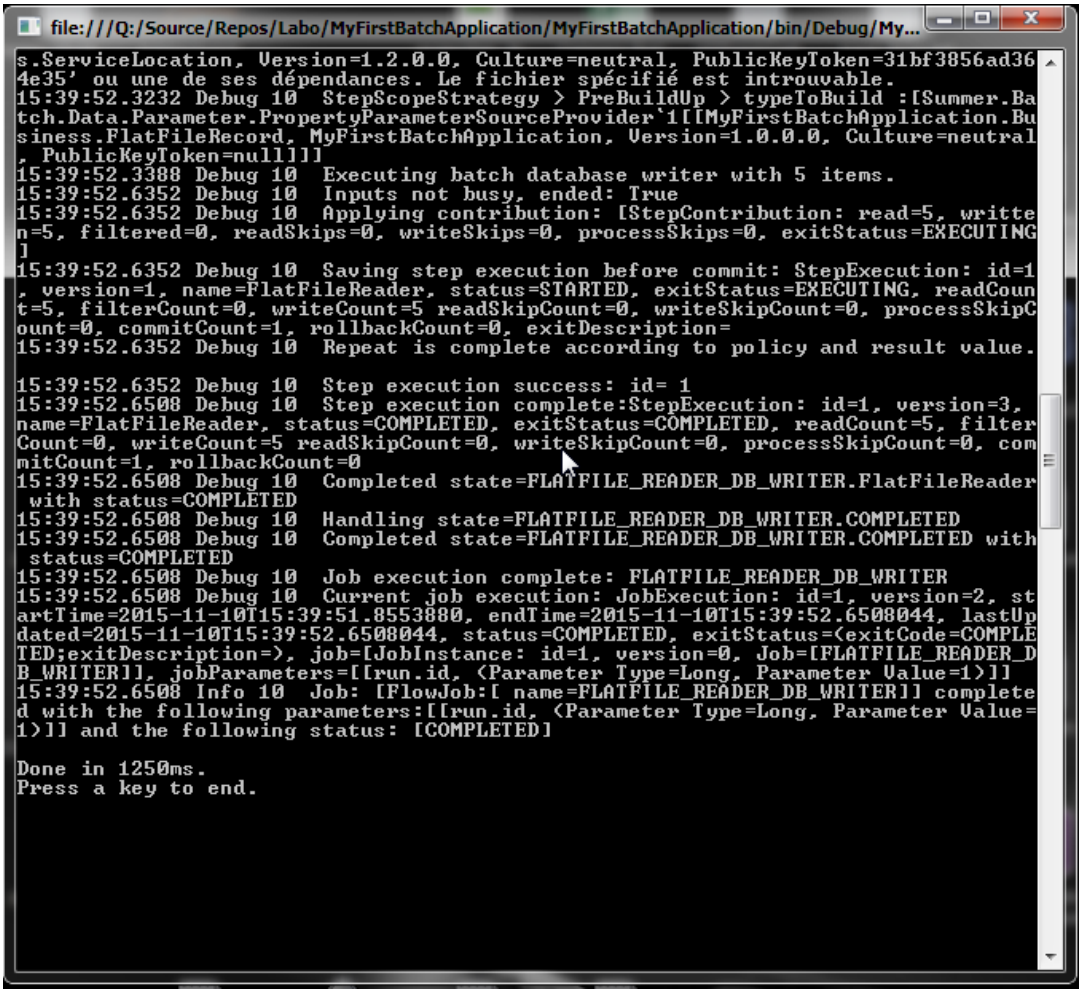
```
1;FlatFile1;FlatFile1;20100101
2;FlatFile2;FlatFile2;19700731
3;FlatFile3;FlatFileDesc3;19690420
4;FlatFile4;FlatFile4;20070928
5;FlatFile5;FlatFileDesc5;20151109
```

Caution

Be sure to select the 'Copy Always' option for the 'Copy to Output Directory' property, in the Properties windows of the FlatFile.txt file, or it won't be visible at run time and job will fail with a `System.InvalidOperationException: Input resource must exist (reader is in 'strict' mode)` exception message.

Build the solution -- stick to Debug mode for now -- (F6), then run the `Program`, using F5. You should see the logs in the execution console, similar to the following screenshot :

Figure 16. Sample batch execution console view



Of course, timings may vary depending on your machine configuration (cpu/disks/ram/OS).

The console view is not very usable for large outputs; The log file (set to be to located at `C:\temp\MyFirstBatchApplicationLog.txt` according to the NLog configuration) can be opened in any text editor to browse comfortably the execution logs.
Make sure the batch performed properly :

- The final log entry shoud state that the job completed with the `COMPLETED` status, like below :

```
17:03:00.3949 Info 10  Job: [FlowJob:[ name=FLATFILE_READER_DB_WRITER]] completed with the
   following parameters:[[run.id, (Parameter Type=Long, Parameter Value=1)]]
   and the following status: [COMPLETED]
```

- Inspect the logs to see if, for the given sample data, reader, processor and writer did their respective work properly :
  - For the reader :

```
18:06:48.3890 Debug 9  Getting input stream for resource: data/input/FlatFile.txt
18:06:48.3890 Debug 9  Starting repeat context.
18:06:48.3890 Debug 9  Repeat operation about to start at count=1
18:06:48.3890 Debug 9  Preparing chunk execution for StepContext: StepContext@87656246
18:06:48.3890 Debug 9  Chunk execution starting: queue size= 0
18:06:48.4046 Debug 9  Starting repeat context.
18:06:48.4202 Debug 9  Repeat operation about to start at count=1
18:06:48.4202 Debug 9  Repeat operation about to start at count=2
18:06:48.4202 Debug 9  Repeat operation about to start at count=3
18:06:48.4202 Debug 9  Repeat operation about to start at count=4
18:06:48.4202 Debug 9  Repeat operation about to start at count=5
18:06:48.4202 Debug 9  Repeat operation about to start at count=6
18:06:48.4202 Debug 9  Repeat is complete according to policy and result value.
```

  - For the processor :

```
18:06:48.4202 Debug 9 (FlatFileRecordProcessor.cs:24) Treating item with code 1
18:06:48.4202 Debug 9 (FlatFileRecordProcessor.cs:29) Missing description for item 1
18:06:48.4202 Debug 9 (FlatFileRecordProcessor.cs:24) Treating item with code 2
18:06:48.4358 Debug 9 (FlatFileRecordProcessor.cs:29) Missing description for item 2
18:06:48.4358 Debug 9 (FlatFileRecordProcessor.cs:24) Treating item with code 3
18:06:48.4358 Debug 9 (FlatFileRecordProcessor.cs:24) Treating item with code 4
18:06:48.4358 Debug 9 (FlatFileRecordProcessor.cs:29) Missing description for item 4
18:06:48.4358 Debug 9 (FlatFileRecordProcessor.cs:24) Treating item with code 5
```

  - For the writer :

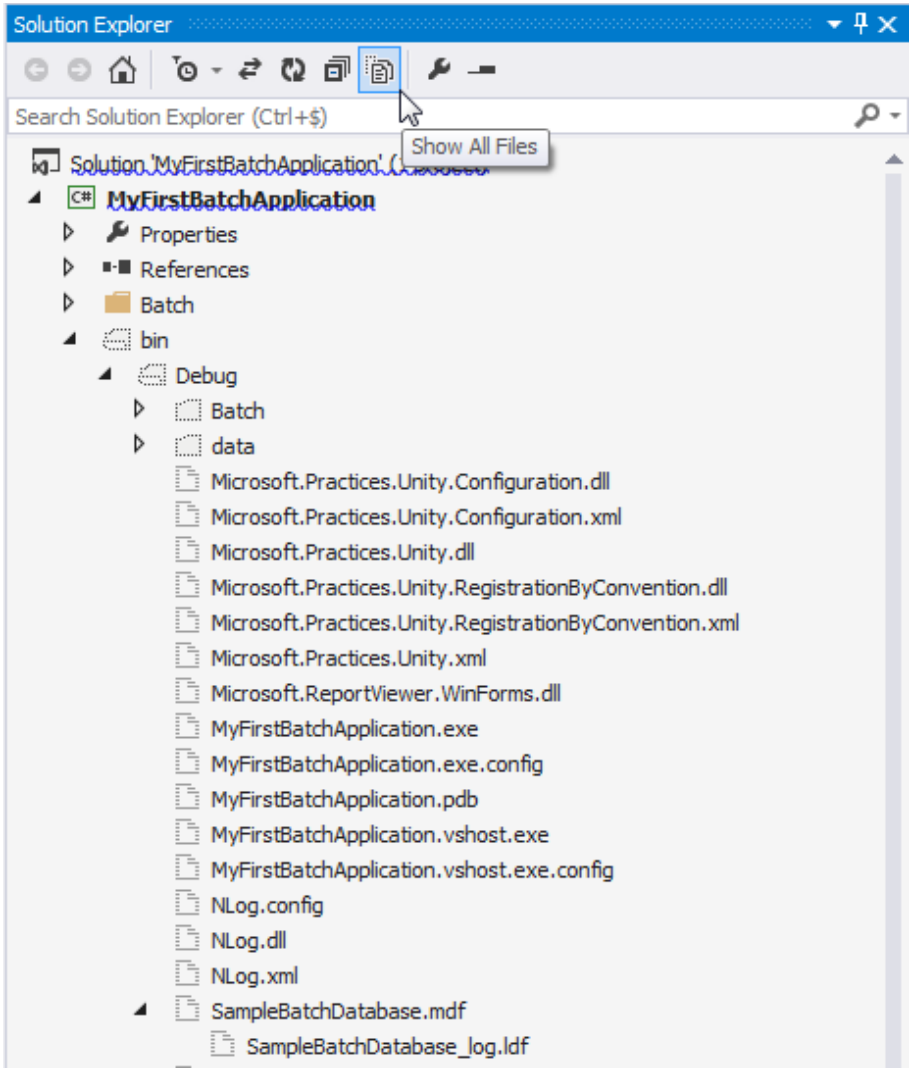
```
18:06:48.7322 Debug 9  Executing batch database writer with 5 items.
18:06:49.1066 Debug 9  Inputs not busy, ended: True
18:06:49.1066 Debug 9  Applying contribution: [StepContribution: read=5, written=5,
filtered=0, readSkips=0, writeSkips=0, processSkips=0, exitStatus=EXECUTING]
```

- Check the database content

Caution

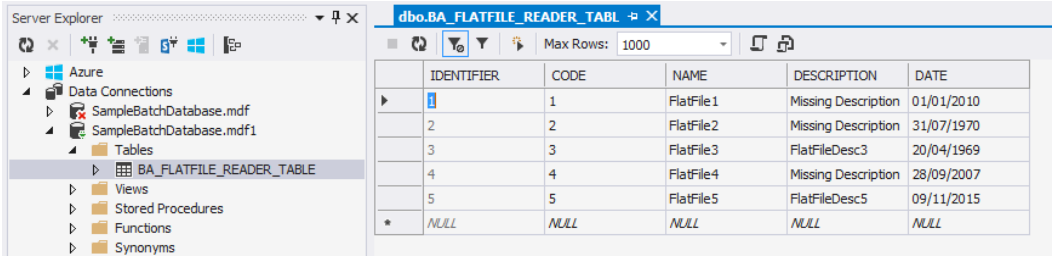
The database that is written to is NOT the one you created in the project. Indeed, the created database SampleBatchDatabase.mdf is being copied to the build directory (whose path depends on the build mode you chose) and this is the one that will be used at runtime. To be able to browse the database content, you must use the 'Show All Files' option, in the Solution Explorer window. In our example, the database is located in the `bin\Debug` folder :

Figure 17. Finding the database in the output directory



For the given data, the content of the BA_FLATFILE_READER_TABLE should be exactly as below

Figure 18. Checking the written table content



Congratulations ! You achieved to write your first Summer Batch project in working order. It only takes 40 lines of code, according to VS Metrics (not including the few lines of xml configuration files).

Going further ...

A more significant test involves dealing with a much larger flat file. Here is a flat file with 10 000 lines, named LargeFlatFile.txt (right click to download the file);

To use it, copy the file into the `data/input` folder, set the 'Copy Always' build instruction (Properties view) and edit the `MyFirstBatchUnityLoader` to point at it as the input file resource:

Switch from :

```
//input file
var inputFileResource = new FileSystemResource("data/input/FlatFile.txt");
```

To :

```
//input file
var inputFileResource = new FileSystemResource("data/input/LargeFlatFile.txt");
```

Rebuild the solution (F6), then run it (F5).
Examine the log file to find out the chunking mechanism evidences :

- The chunk size has been set to 1000; items are first being read

```
10:54:36.5543 Debug 10  Getting input stream for resource: data/input/LargeFlatFile.txt
10:54:36.5543 Debug 10  Starting repeat context.
10:54:36.5543 Debug 10  Repeat operation about to start at count=1
10:54:36.5543 Debug 10  Preparing chunk execution for StepContext: StepContext@87656246
10:54:36.5543 Debug 10  Chunk execution starting: queue size= 0
10:54:36.5855 Debug 10  Starting repeat context.
10:54:36.5855 Debug 10  Repeat operation about to start at count=1
10:54:36.5855 Debug 10  Repeat operation about to start at count=2
10:54:36.5855 Debug 10  Repeat operation about to start at count=3
10:54:36.5855 Debug 10  Repeat operation about to start at count=4
...
...
10:54:36.8351 Debug 10  Repeat operation about to start at count=997
10:54:36.8351 Debug 10  Repeat operation about to start at count=998
10:54:36.8351 Debug 10  Repeat operation about to start at count=999
10:54:36.8351 Debug 10  Repeat operation about to start at count=1000
10:54:36.8351 Debug 10  Repeat is complete according to policy and result value.
```

- Then the processor does its work on the 1000 read items :

```
10:54:36.8507 Debug 10 (FlatFileRecordProcessor.cs:24) Treating item with code 1
10:54:36.8507 Debug 10 (FlatFileRecordProcessor.cs:29) Missing description for item 1
10:54:36.8507 Debug 10 (FlatFileRecordProcessor.cs:24) Treating item with code 2
10:54:36.8507 Debug 10 (FlatFileRecordProcessor.cs:29) Missing description for item 2
...
...
10:54:37.3500 Debug 10 (FlatFileRecordProcessor.cs:24) Treating item with code 998
10:54:37.3500 Debug 10 (FlatFileRecordProcessor.cs:29) Missing description for item 998
10:54:37.3500 Debug 10 (FlatFileRecordProcessor.cs:24) Treating item with code 999
10:54:37.3500 Debug 10 (FlatFileRecordProcessor.cs:24) Treating item with code 1000
```

- Then the writer dumps all the treated items to the database :

```
10:54:37.6464 Debug 10  Executing batch database writer with 1000 items.
10:54:38.0052 Debug 10  Inputs not busy, ended: False
10:54:38.0052 Debug 10  Applying contribution: [StepContribution: read=1000, written=1000,
filtered=0, readSkips=0, writeSkips=0, processSkips=0, exitStatus=EXECUTING]
10:54:38.0052 Debug 10  Saving step execution before commit: StepExecution: id=1, version=1,
name=FlatFileReader, status=STARTED, exitStatus=EXECUTING, readCount=1000, filterCount=0,
writeCount=1000 readSkipCount=0, writeSkipCount=0, processSkipCount=0, commitCount=1,
rollbackCount=0, exitDescription=
```

- and since all data has not been read from the file yet, another repeat operation is being launched to treat the next chunk of data

```
10:54:38.0052 Debug 10  Repeat operation about to start at count=2
10:54:38.0052 Debug 10  Preparing chunk execution for StepContext: StepContext@87656246
10:54:38.0052 Debug 10  Chunk execution starting: queue size= 0
10:54:38.0052 Debug 10  Starting repeat context.
10:54:38.0052 Debug 10  Repeat operation about to start at count=1
10:54:38.0052 Debug 10  Repeat operation about to start at count=2
...
```

- This pattern is repeated until no more data can be read from the input file. The number of transactions commits is being incremented each time the writer is being called ; Also note the `readCount` and `writeCount` counters being incremented with the going chunks :

```
10:54:38.7696 Debug 10  Executing batch database writer with 1000 items.
10:54:38.8320 Debug 10  Inputs not busy, ended: False
10:54:38.8320 Debug 10  Applying contribution: [StepContribution: read=1000, written=1000,
filtered=0, readSkips=0, writeSkips=0, processSkips=0, exitStatus=EXECUTING]
10:54:38.8320 Debug 10  Saving step execution before commit: StepExecution: id=1, version=2,
name=FlatFileReader, status=STARTED, exitStatus=EXECUTING, readCount=2000, filterCount=0,
writeCount=2000 readSkipCount=0, writeSkipCount=0, processSkipCount=0, commitCount=2,
rollbackCount=0, exitDescription=
...
...
10:54:45.2906 Debug 10  Executing batch database writer with 1000 items.
10:54:45.3530 Debug 10  Inputs not busy, ended: False
10:54:45.3530 Debug 10  Applying contribution: [StepContribution: read=1000, written=1000,
filtered=0, readSkips=0, writeSkips=0, processSkips=0, exitStatus=EXECUTING]
10:54:45.3530 Debug 10  Saving step execution before commit: StepExecution: id=1,
version=9, name=FlatFileReader,
status=STARTED, exitStatus=EXECUTING, readCount=9000, filterCount=0, writeCount=9000
readSkipCount=0, writeSkipCount=0,
processSkipCount=0, commitCount=9, rollbackCount=0, exitDescription=
10:54:45.3530 Debug 10  Repeat operation about to start at count=10
...
10:54:46.1798 Debug 10  Executing batch database writer with 1000 items.
10:54:46.2578 Debug 10  Inputs not busy, ended: False
10:54:46.2578 Debug 10  Applying contribution: [StepContribution: read=1000, written=1000,
filtered=0, readSkips=0, writeSkips=0, processSkips=0, exitStatus=EXECUTING]
10:54:46.2578 Debug 10  Saving step execution before commit: StepExecution: id=1,
version=10, name=FlatFileReader,
status=STARTED, exitStatus=EXECUTING, readCount=10000, filterCount=0, writeCount=10000
readSkipCount=0, writeSkipCount=0,
processSkipCount=0, commitCount=10, rollbackCount=0, exitDescription=
10:54:46.2578 Debug 10  Repeat operation about to start at count=11
10:54:46.2578 Debug 10  Preparing chunk execution for StepContext: StepContext@87656246
10:54:46.2578 Debug 10  Chunk execution starting: queue size= 0
10:54:46.2578 Debug 10  Starting repeat context.
10:54:46.2578 Debug 10  Repeat operation about to start at count=1
10:54:46.2578 Debug 10  Repeat is complete according to policy and result value.
10:54:46.2578 Debug 10  Inputs not busy, ended: True
```

Note

The batch performance is bound to the build mode and the log level. Try playing with these and watch the impact on the batch performance.

Here are sample results from my desktop, using the LargeFlatFile.txt as input file:

Build Mode	Log Level	Timing (in ms)
DEBUG	Debug	10 206
DEBUG	Info	1 944
RELEASE	Debug	9 827
RELEASE	Info	1 799

Caution

As expected, RELEASE mode is slightly better than DEBUG mode but what really impacts the batch performance is the log level.

The Debug log level should only be used while developping the batch and *NOT* in a production environment.

Now it's time to build your own batch projects. To dig into Summer Batch API, please consider browsing the following resources :

- The HTML Summer Batch Reference Guide (single page)
- The HTML Summer Batch Reference Guide (multi-pages)
- The HTML Summer Batch API doc.
- The Summer Batch GitHub repository

Index