Devshed.Csv 1.2 documentation

Draft 2014-8-10 13:53

Introduction

The Devshed Tools have been designed to accumulate common functionality, that provide as much compile-time support and least strong coupling with your code. By that we mean inheritance or attribute usage.

The Devshed CSV assembly provides basic CSV read and write functionality by exposing a CsvReader and CsvWriter class. Using a CsvDefinition object you can specify how to read and write any type of array.

Noticeable or breaking changes in 1.2 since 1.1

#1658 Convenience methods string Build() and byte[] BuildBytes() have been removed. Only *void Build(Stream)* and *Stream Build()* are available on the CsvBuilder class.

#1659 Introduced the static CsvReader class for convenience and consistency with the CsvBuilder class.

#1661 Added 'ElementDelimiter' to the CsvDefinition<> to specify the separating element character.

#1664 Renamed Each method in the LinqExtensions to ForEach() for clarity.

Writing with the CsvWriter

CsvWriter is a static class that does all the object wiring for you. It has two static methods for writing an existing stream and creating a new one. A CsvDefintion<T> and a collection are needed to write the CSV content.

Defining the type mapping

The following example demonstrates definition of the property mappings to the UserView object. Nothing else is needed.

```
var definition = new CsvDefinition<UserView>(
    new NumberCsvColumn<UserView >(e => e.Id),
    new TextCsvColumn<UserView >(e => e.Name),
    new BooleanCsvColumn<UserView >(e => e.IsActive));
```

After that, create an array of the UserView type, which could be database records as well:

```
var users = new UserView[]
{
      new UserView
            {
                  Id = 1,
                  Name = “John”,
                  IsActive = true
            },

      new UserView
            {
                  Id = 2,
                  Name = “Marry”,
                  IsActive = false
            }
}
```

Using a the CsvWriter with a StreamWriter you can write directly to a file:

```
using(var stream = new FileStream("C:\\Test.CSV", FileMode.Create))
{
    CsvWriter.Write(stream, definition, users);
}
```

Extension methods allow using the definition directly, like this:

```
using(var stream = new FileStream("C:\\Test.CSV", FileMode.Create))
{
    definition.WriteStream(stream, users);
}
```

Reading CSV with the CsvReader

Like the CsvWriter, the CsvReader follows the same principle by using a definition and a source stream; it reads and materializes the contents to an array:

```
var definition = new CsvDefinition<UserView>(
    new NumberCsvColumn<UserView >(e => e.Id),
    new TextCsvColumn<UserView >(e => e.Name),
    new BooleanCsvColumn<UserView >(e => e.IsActive));
```

Use as stream for content and present both to the reader:

```
using (var reader = new FileStream("C:\\Test.CSV", FileMode.Open))
{
    var users = CsvReader.Read<UserView>(stream, definition);
}
```

Like that, 'users' now contains an array of UserViews objects. A reading extension is also available on the definition:

```
using (var reader = new FileStream("C:\\Test.CSV", FileMode.Open))
{
    var users = definition.ReadStream<UserView>(stream);
}
```

Additional CsvDefinition<> options

The CsvDefintion<TRow> has the following three additional properties:

FirstRowContainsHeaders (default false)

This option is for both writing and reading. The first line will contain the header names of the defined columns. By default the names will be the reflected property names, which can be overridden.

When a custom header name is required, it can be overruled with the HeaderName property. More on this in the 'CsvColumn definitions' section.

When reading a file with this option on, the reader expects the first line to be filled with header names as quoted strings. The names must match either the reflected name or the custom name. If no header names are specified, the reader works index based.

RemoveNewLineCharacters (default false)

When enabled all written CSV text fields will be stripped of new line characters. For writing CSV only. NOTE; header names will be stripped from newline characters anyway.

ElementDelimiter (default ";")

The character that separates the CSV elements for reading and writing CSV content.

Column definitions

In order to configure your CSV definition, several column types are available. All columns expose a Format and HeaderName property in order to overwrite the default behavior. Note that some of the values passing through the Format expression can be null. For example the DecimalCsvColumn.

The following examples are written with their default values.

TextCsvColumn<TSource>

Accepts strings and can remove new line characters.

```
new TextCsvColumn<TSource>(e => e.TProperty)
{
    Format = value => value + "_ADD_THIS",
    HeaderName = "Overruled Name",
    ForceExcelTextCell = true
}
```

ForceExcelTextCell

Writes the text value like ="VALUE" instead of "VALUE" to force Excel to parse it as text otherwise textual numbers will still be treated as number.

CurrencyCsvColumn<TSource>

Accepts nullable decimal and formats them to the current culture currency of the thread.

```
new CurrencyCsvColumn<TSource>(e => e.TProperty)
{
    Format = value => string.Format("{0:c2}", value ?? 0),
    HeaderName = "Overruled Name"
}
```

NumberCsvColumn<TSource>

Accepts nullable int and formats them to the current culture currency of the thread.

```
new NumberCsvColumn<TSource>(e => e.TProperty)
{
    Format = value => string.Format("{0:0.00}", value ?? 0),
    HeaderName = "Overruled Name"
}
```

DecimalCsvColumn<TSource>

Accepts nullable decimal values and formats them to the current culture currency of the thread.

```
new CurrencyCsvColumn<TSource>(e => e.TProperty)
{
    Format = value => string.Format("{0:c4}", value ?? 0),
    HeaderName = "Overruled Name",
    Format = (number, formatter) => number != null
                ? number.Value.ToString(formatter)
                : string.Empty,
}
```

DateCsvColumn<TSource>

Accepts nullable DateTime values and formats them to a short date format.

```
new DateCsvColumn<TSource>(e => e.TProperty)
{
    Format = value => string.Format(“{0:c4}”, value ?? 0),
    HeaderName = “Overruled Name”,
    Format = e => e != null
         ? e.Value.ToShortDateString()
         : string.Empty
}
```

TimeCsvColumn<TSource>

Accepts nullable decimal values and formats them to the current culture currency of the thread.

```
new TimeCsvColumn<TSource>(e => e.TProperty)
{
    Format = value => string.Format(“{0:c4}”, value ?? 0),
    HeaderName = “Overruled Name” ”,
    Format = e => e != null
         ? string.Format("{0:00}:{1:00}", e.Value.Hours, e.Value.Minutes)
         : string.Empty
}
```

BooleanCsvColumn<TSource>

Accepts boolean values and formats them to string.

```
new TimeCsvColumn<TSource>(e => e.TProperty)
{
    Format = value => string.Format(“{0:c4}”, value ?? 0),
    HeaderName = “Overruled Name”,
    Format = e => e.ToString()
}
```

Special columns

ObjectCsvColumn<TSource>

Accepts any type of value and formatting can be handled as you please.

```
new ObjectCsvColumn<TSource>(e => e.TProperty)
{
    Format = value => string.Format(“{0:c4}”, value ?? 0),
    HeaderName = “Overruled Name”,
    Format = e => e.ToString()
}
```

ArrayCsvColumn<TSource, TArray>

This column type allows to write the content of an array be written in a single element. It accepts a mapping to a property of an array type TArray.

```
new ArrayCsvColumn<TSource, int>(e => e.TProperty)
{
    Format = value => string.Format("{0:c4}", value ?? 0),
    HeaderName = "Overruled Name",
    Format = e => e.ToString()
}
```

The Format expression is executed with each element in the array. In the CSV the values will be written comma separated like "1,2,3". For example:

"John";1,2,3;"Street"

This does not allow strings to be written either newlines or delimiter characters like ";".

CompositeCsvColumn<TSource, TValue>

This column was introduced to solve problem of dynamic fields, when generating financial exports that require for example tax groups that cannot be predefined in the compiled definition.

Unavoidably this requires the use of a CompositeColumnValue<,> object from the assembly. A KeyValuePair<string, string> that was used in prior versions was considered as too unspecific for the purpose.

```
new CompositeCsvColumn<TSource, string>(
    e => e.Collection, "Header1", "Header2")
{
    Format = value => string.Format("{0:c4}", value ?? 0),
    HeaderName = "Overruled Name",
    Format = e => e.ToString()
}

var rows = new [] {
    new Model {
        Name = "John",
        Collection = new []
        {
            new CompositeColumnValue <string>("Header 1", "Value 1"),
            new CompositeColumnValue <string>("Header 2", "Value 2")
        }
    }
}
```

In the CSV the values will be written as separate elements, each with their own header name, like this:

"Name", "Header1", "Header2"
"John";"Value 1";"Value 2"

Note that the values of the composition will be treated as strings.