Bindy

Bindy

The goal of this component is to allow the parsing/binding of non-structured data (or to be more precise non-XML data)
to/from Java Beans that have binding mappings defined with annotations. Using Bindy, you can bind data from sources such as :

- CSV records,
- Fixed-length records,
- FIX messages,
- or almost any other non-structured data

to one or many Plain Old Java Object (POJO). Bindy converts the data according to the type of the java property. POJOs can be linked together with one-to-many relationships available in some cases. Moreover, for data type like Date, Double, Float, Integer, Short, Long and BigDecimal, you can provide the pattern to apply during the formatting of the property.

For the BigDecimal numbers, you can also define the precision and the decimal or grouping separators.

Type	Format Type	Pattern example	Link
Date	DateFormat	"dd-MM-yyyy"	http://java.sun.com/j2se/1.5.0/docs/api/java/text/SimpleDateFormat.html
Decimal*	Decimalformat	"##.###.###"	http://java.sun.com/j2se/1.5.0/docs/api/java/text/DecimalFormat.html

Decimal* = Double, Integer, Float, Short, Long

Format supported

This first release only support comma separated values fields and key value pair fields (e.g. : FIX messages).

To work with camel-bindy, you must first define your model in a package (e.g. com.acme.model) and for each model class (e.g. Order, Client, Instrument, ...) add the required annotations (described hereafter) to the Class or field.

Multiple models

If you use multiple models, each model has to be placed in it's own package to prevent unpredictable results.

From **Camel 2.16** onwards this is no longer the case, as you can safely have multiple models in the same package, as you configure bindy using class names instead of package names now.

Annotations

The annotations created allow to map different concept of your model to the POJO like :

- Type of record (csv, key value pair (e.g. FIX message), fixed length ...),
- Link (to link object in another object),
- DataField and their properties (int, type, ...),
- KeyValuePairField (for key = value format like we have in FIX financial messages),
- Section (to identify header, body and footer section),
- OneToMany

This section will describe them :

1. CsvRecord

The CsvRecord annotation is used to identified the root class of the model. It represents a record = a line of a CSV file and can be linked to several children model classes.

Annotation name	Record type	Level
CsvRecord	csv	Class

Parameter name	type	Info
separator	string	mandatory - can be ',' or ';' or 'anything'. This value is interpreted as a regular expression. If you want to use a sign which has a special meaning in regular expressions, e.g. the '|' sign, than you have to mask it, like ' |'
skipFirstLine	boolean	optional - default value = false - allow to skip the first line of the CSV file

crlf	string	optional - possible values = WINDOWS,UNIX,MAC, or custom; default value = WINDOWS - allow to define the carriage return character to use. If you specify a value other than the three listed before, the value you enter (custom) will be used as the CRLF character(s)
generateHeaderColumns	boolean	optional - default value = false - uses to generate the header columns of the CSV generates
autospanLine	boolean	**Camel 2.13/2.12.2:** optional - default value = false - if enabled then the last column is auto spanned to end of line, for example if its a comment, etc this allows the line to contain all characters, also the delimiter char.
isOrdered	boolean	optional - default value = false - allow to change the order of the fields when CSV is generated
quote	String	**Camel 2.8.3/2.9:** option - allow to specify a quote character of the fields when CSV is generated
		This annotation is associated to the root class of the model and must be declared one time.
quoting	boolean	**Camel 2.11:**optional - default value = false - Indicate if the values must be quoted when marshaling when CSV is generated.

case 1 : separator = ','

The separator used to segregate the fields in the CSV record is ',' :

10, J, Pauline, M, XD12345678, Fortis Dynamic 15/15, 2500, USD,08-01-2009

```
@CsvRecord( separator = "," )
public Class Order {
...
}
```

case 2 : separator = ';'

Compare to the previous case, the separator here is ';' instead of ',' :

10; J; Pauline; M; XD12345678; Fortis Dynamic 15/15; 2500; USD; 08-01-2009

```
@CsvRecord( separator = ";" )
public Class Order {
...
}
```

case 3 : separator = '|'

Compare to the previous case, the separator here is '|' instead of ';' :

10| J| Pauline| M| XD12345678| Fortis Dynamic 15/15| 2500| USD| 08-01-2009

```
@CsvRecord( separator = "\\|" )
public Class Order {
...
}
```

case 4 : separator = '\",\"'
Applies for Camel 2.8.2 or older

When the field to be parsed of the CSV record contains ',' or ';' which is also used as separator, we whould find another strategy
to tell camel bindy how to handle this case. To define the field containing the data with a comma, you will use simple or double quotes
as delimiter (e.g : '10', 'Street 10, NY', 'USA' or "10", "Street 10, NY", "USA").
Remark : In this case, the first and last character of the line which are a simple or double quotes will removed by bindy

"10","J","Pauline"," M","XD12345678","Fortis Dynamic 15,15" 2500","USD","08-01-2009"

```
@CsvRecord( separator = "\",\"" )
public Class Order {
...
}
```

From **Camel 2.8.3/2.9 or never** bindy will automatic detect if the record is enclosed with either single or double quotes and automatic remove those quotes when unmarshalling from CSV to Object. Therefore do **not** include the quotes in the separator, but simple do as below:

"10","J","Pauline"," M","XD12345678","Fortis Dynamic 15,15" 2500","USD","08-01-2009"

```
@CsvRecord( separator = "," )
public Class Order {
...
}
```

Notice that if you want to marshal from Object to CSV and use quotes, then you need to specify which quote character to use, using the `quote` attribute on the @CsvRecord as shown below:

```
@CsvRecord( separator = ",", quote = "\"" )
public Class Order {
...
}
```

case 5 : separator & skipfirstline

The feature is interesting when the client wants to have in the first line of the file, the name of the data fields :

order id, client id, first name, last name, isin code, instrument name, quantity, currency, date

To inform bindy that this first line must be skipped during the parsing process, then we use the attribute :

```
@CsvRecord(separator = ",", skipFirstLine = true)
public Class Order {
...
}
```

case 6 : generateHeaderColumns

To add at the first line of the CSV generated, the attribute generateHeaderColumns must be set to true in the annotation like this :

```
@CsvRecord( generateHeaderColumns = true )
public Class Order {
...
}
```

As a result, Bindy during the unmarshaling process will generate CSV like this :

order id, client id, first name, last name, isin code, instrument name, quantity, currency, date
10, J, Pauline, M, XD12345678, Fortis Dynamic 15/15, 2500, USD,08-01-2009

case 7 : carriage return

If the platform where camel-bindy will run is not Windows but Macintosh or Unix, than you can change the crlf property like this. Three values are available : WINDOWS, UNIX or MAC

```
@CsvRecord(separator = ",", crlf="MAC")
public Class Order {
...
}
```

Additionally, if for some reason you need to add a different line ending character, you can opt to specify it using the crlf parameter. In the following example, we can end the line with a comma followed by the newline character:

```
@CsvRecord(separator = ",", crlf=",\n")
public Class Order {
...
}
```

case 8 : isOrdered

Sometimes, the order to follow during the creation of the CSV record from the model is different from the order used during the parsing. Then, in this case, we can use the attribute isOrdered = true to indicate this in combination with attribute 'position' of the DataField annotation.

```
@CsvRecord(isOrdered = true)
public Class Order {

   @DataField(pos = 1, position = 11)
   private int orderNr;

   @DataField(pos = 2, position = 10)
   private String clientNr;

...
}
```

Remark : pos is used to parse the file, stream while positions is used to generate the CSV

2. Link

The link annotation will allow to link objects together.

Annotation name	Record type	Level
Link	all	Class & Property

Parameter name	type	Info
linkType	LinkType	optional - by default the value is LinkType.oneToOne - so you are not obliged to mention it
		Only one-to-one relation is allowed.

e.g : If the model Class Client is linked to the Order class, then use annotation Link in the Order class like this :

Property Link

```
@CsvRecord(separator = ",")
public class Order {

    @DataField(pos = 1)
    private int orderNr;

    @Link
    private Client client;
...
```

AND for the class Client :

Class Link

```
@Link
public class Client {
...
}
```

3. DataField

The DataField annotation defines the property of the field. Each datafield is identified by its position in the record, a type (string, int, date, ...) and optionally of a pattern

Annotation name	Record type	Level
DataField	all	Property

Parameter name	type	Info

pos	int	mandatory - The **input** position of the field. digit number starting from 1 to ... - See the position parameter.
pattern	string	optional - default value = "" - will be used to format Decimal, Date, ...
length	int	optional - represents the length of the field for fixed length format
precision	int	optional - represents the precision to be used when the Decimal number will be formatted/parsed
pattern	string	optional - default value = "" - is used by the Java formatter (SimpleDateFormat by example) to format/validate data. If using pattern, then setting locale on bindy data format is recommended. Either set to a known locale such as "us" or use "default" to use platform default locale. Notice that "default" requires Camel 2.14/2.13.3/2.12.5.
position	int	optional - must be used when the position of the field in the CSV generated (output message) must be different compare to input position (pos). See the pos parameter.
required	boolean	optional - default value = "false"
trim	boolean	optional - default value = "false"
defaultValue	string	**Camel 2.10:** optional - default value = "" - defines the field's default value when the respective CSV field is empty/not available
impliedDecimalSeparator	boolean	**Camel 2.11:** optional - default value = "false" - Indicates if there is a decimal point implied at a specified location
lengthPos	int	**Camel 2.11**: optional - can be used to identify a data field in a fixed-length record that defines the fixed length for this field
delimiter	string	**Camel 2.11:** optional - can be used to demarcate the end of a variable-length field within a fixed-length record

case 1 : pos

This parameter/attribute represents the position of the field in the csv record

Position

```
@CsvRecord(separator = ",")
public class Order {

    @DataField(pos = 1)
    private int orderNr;

    @DataField(pos = 5)
    private String isinCode;

...
}
```

As you can see in this example the position starts at '1' but continues at '5' in the class Order. The numbers from '2' to '4' are defined in the class Client (see here after).

Position continues in another model class

```
public class Client {

    @DataField(pos = 2)
    private String clientNr;

    @DataField(pos = 3)
    private String firstName;

    @DataField(pos = 4)
    private String lastName;
...
}
```

case 2 : pattern

The pattern allows to enrich or validates the format of your data

Pattern

```
@CsvRecord(separator = ",")
public class Order {

    @DataField(pos = 1)
    private int orderNr;

    @DataField(pos = 5)
    private String isinCode;

    @DataField(name = "Name", pos = 6)
    private String instrumentName;

    @DataField(pos = 7, precision = 2)
    private BigDecimal amount;

    @DataField(pos = 8)
    private String currency;

    @DataField(pos = 9, pattern = "dd-MM-yyyy") -- pattern used during parsing or when the date is created
    private Date orderDate;
...
}
```

case 3 : precision

The precision is helpful when you want to define the decimal part of your number

Precision

```
@CsvRecord(separator = ",")
public class Order {

    @DataField(pos = 1)
    private int orderNr;

    @Link
    private Client client;

    @DataField(pos = 5)
    private String isinCode;

    @DataField(name = "Name", pos = 6)
    private String instrumentName;

    @DataField(pos = 7, precision = 2) -- precision
    private BigDecimal amount;

    @DataField(pos = 8)
    private String currency;

    @DataField(pos = 9, pattern = "dd-MM-yyyy")
    private Date orderDate;
...
}
```

case 4 : Position is different in output

The position attribute will inform bindy how to place the field in the CSV record generated. By default, the position used corresponds to the position defined with the attribute 'pos'. If the position is different (that means that we have an asymetric processus comparing marshaling from unmarshaling) than we can use 'position' to indicate this.

Here is an example

Position is different in output

```
@CsvRecord(separator = ",")
public class Order {
@CsvRecord(separator = ",", isOrdered = true)
public class Order {

    // Positions of the fields start from 1 and not from 0

    @DataField(pos = 1, position = 11)
    private int orderNr;

    @DataField(pos = 2, position = 10)
    private String clientNr;

    @DataField(pos = 3, position = 9)
    private String firstName;

    @DataField(pos = 4, position = 8)
    private String lastName;

    @DataField(pos = 5, position = 7)
    private String instrumentCode;

    @DataField(pos = 6, position = 6)
    private String instrumentNumber;
...
}
```

This attribute of the annotation @DataField must be used in combination with attribute isOrdered = true of the annotation @CsvRecord

case 5 : required

If a field is mandatory, simply use the attribute 'required' setted to true

Required

```
@CsvRecord(separator = ",")
public class Order {

    @DataField(pos = 1)
    private int orderNr;

    @DataField(pos = 2, required = true)
    private String clientNr;

    @DataField(pos = 3, required = true)
    private String firstName;

    @DataField(pos = 4, required = true)
    private String lastName;
...
}
```

If this field is not present in the record, than an error will be raised by the parser with the following information :

Some fields are missing (optional or mandatory), line :

case 6 : trim

If a field has leading and/or trailing spaces which should be removed before they are processed, simply use the attribute 'trim' setted to true

Trim

```
@CsvRecord(separator = ",")
public class Order {

    @DataField(pos = 1, trim = true)
    private int orderNr;

    @DataField(pos = 2, trim = true)
    private Integer clientNr;

    @DataField(pos = 3, required = true)
    private String firstName;

    @DataField(pos = 4)
    private String lastName;
...
}
```

case 7 : defaultValue

If a field is not defined then uses the value indicated by the defaultValue attribute

Default value

```
@CsvRecord(separator = ",")
public class Order {

    @DataField(pos = 1)
    private int orderNr;

    @DataField(pos = 2)
    private Integer clientNr;

    @DataField(pos = 3, required = true)
    private String firstName;

    @DataField(pos = 4, defaultValue = "Barin")
    private String lastName;
...
}
```

This attribute is only applicable to optional fields.

4. FixedLengthRecord

The FixedLengthRecord annotation is used to identified the root class of the model. It represents a record = a line of a file/message containing data fixed length formatted and can be linked to several children model classes. This format is a bit particular beause data of a field can be aligned to the right or to the left.
When the size of the data does not fill completely the length of the field, we can then add 'padd' characters.

Annotation name	Record type	Level
FixedLengthRecord	fixed	Class

Parameter name	type	Info
crlf	string	optional - possible values = WINDOWS,UNIX,MAC, or custom; default value = WINDOWS - allow to define the carriage return character to use. If you specify a value other than the three listed before, the value you enter (custom) will be used as the CRLF character(s)
paddingChar	char	mandatory - default value = ' '
length	int	mandatory = size of the fixed length record

hasHeader	boolean	**Camel 2.11** - optional - Indicates that the record(s) of this type may be preceded by a single header record at the beginning of the file / stream
hasFooter	boolean	**Camel 2.11** - optional - Indicates that the record(s) of this type may be followed by a single footer record at the end of the file / stream
skipHeader	boolean	**Camel 2.11** - optional - Configures the data format to skip marshalling / unmarshalling of the header record. Configure this parameter on the primary record (e.g., not the header or footer).
skipFooter	boolean	**Camel 2.11** - optional - Configures the data format to skip marshalling / unmarshalling of the footer record Configure this parameter on the primary record (e.g., not the header or footer)..
isHeader	boolean	**Camel 2.11** - optional - Identifies this FixedLengthRecord as a header record
isFooter	boolean	**Camel 2.11** - optional - Identifies this FixedLengthRecords as a footer record
ignoreTrailingChars	boolean	**Camel 2.11.1** - optional - Indicates that characters beyond the last mapped filed can be ignored when unmarshalling / parsing.
		This annotation is associated to the root class of the model and must be declared one time.

The hasHeader/hasFooter parameters are mutually exclusive with isHeader/isFooter. A record may not be both a header/footer and a primary fixed-length record.

case 1 : Simple fixed length record

This simple example shows how to design the model to parse/format a fixed message

10A9PaulineMISINXD12345678BUYShare2500.45USD01-08-2009

Fixed-simple

```
@FixedLengthRecord(length=54, paddingChar=' ')
 public static class Order {

     @DataField(pos = 1, length=2)
     private int orderNr;

     @DataField(pos = 3, length=2)
     private String clientNr;

     @DataField(pos = 5, length=7)
     private String firstName;

     @DataField(pos = 12, length=1, align="L")
     private String lastName;

     @DataField(pos = 13, length=4)
     private String instrumentCode;

     @DataField(pos = 17, length=10)
     private String instrumentNumber;

     @DataField(pos = 27, length=3)
     private String orderType;

     @DataField(pos = 30, length=5)
     private String instrumentType;

     @DataField(pos = 35, precision = 2, length=7)
     private BigDecimal amount;

     @DataField(pos = 42, length=3)
     private String currency;

     @DataField(pos = 45, length=10, pattern = "dd-MM-yyyy")
     private Date orderDate;
     ...
```

case 2 : Fixed length record with alignment and padding

This more elaborated example show how to define the alignment for a field and how to assign a padding character which is ' ' here"

10A9 PaulineM ISINXD12345678BUYShare2500.45USD01-08-2009

Fixed-padding-align

```
@FixedLengthRecord(length=60, paddingChar=' ')
 public static class Order {

     @DataField(pos = 1, length=2)
     private int orderNr;

     @DataField(pos = 3, length=2)
     private String clientNr;

     @DataField(pos = 5, length=9)
     private String firstName;

     @DataField(pos = 14, length=5, align="L")   // align text to the LEFT zone of the block
     private String lastName;

     @DataField(pos = 19, length=4)
     private String instrumentCode;

     @DataField(pos = 23, length=10)
     private String instrumentNumber;

     @DataField(pos = 33, length=3)
     private String orderType;

     @DataField(pos = 36, length=5)
     private String instrumentType;

     @DataField(pos = 41, precision = 2, length=7)
     private BigDecimal amount;

     @DataField(pos = 48, length=3)
     private String currency;

     @DataField(pos = 51, length=10, pattern = "dd-MM-yyyy")
     private Date orderDate;
     ...
```

case 3 : Field padding

Sometimes, the default padding defined for record cannnot be applied to the field as we have a number format where we would like to padd with '0' instead of ' '. In this case, you can use in the model the attribute paddingField to set this value.

10A9 PaulineM ISINXD12345678BUYShare000002500.45USD01-08-2009

Fixed-padding-field

```
@FixedLengthRecord(length = 65, paddingChar = ' ')
public static class Order {

    @DataField(pos = 1, length = 2)
    private int orderNr;

    @DataField(pos = 3, length = 2)
    private String clientNr;

    @DataField(pos = 5, length = 9)
    private String firstName;

    @DataField(pos = 14, length = 5, align = "L")
    private String lastName;

    @DataField(pos = 19, length = 4)
    private String instrumentCode;

    @DataField(pos = 23, length = 10)
    private String instrumentNumber;

    @DataField(pos = 33, length = 3)
    private String orderType;

    @DataField(pos = 36, length = 5)
    private String instrumentType;

    @DataField(pos = 41, precision = 2, length = 12, paddingChar = '0')
    private BigDecimal amount;

    @DataField(pos = 53, length = 3)
    private String currency;

    @DataField(pos = 56, length = 10, pattern = "dd-MM-yyyy")
    private Date orderDate;
    ...
```

case 4: Fixed length record with delimiter

Fixed-length records sometimes have delimited content within the record. The firstName and lastName fields are delimited with the '^' character in the following example:

10A9Pauline^M^ISINXD12345678BUYShare000002500.45USD01-08-2009

Fixed-delimited

```
@FixedLengthRecord()
public static class Order {

    @DataField(pos = 1, length = 2)
    private int orderNr;

    @DataField(pos = 2, length = 2)
    private String clientNr;

    @DataField(pos = 3, delimiter = "^")
    private String firstName;

    @DataField(pos = 4, delimiter = "^")
    private String lastName;

    @DataField(pos = 5, length = 4)
    private String instrumentCode;

    @DataField(pos = 6, length = 10)
    private String instrumentNumber;

    @DataField(pos = 7, length = 3)
    private String orderType;

    @DataField(pos = 8, length = 5)
    private String instrumentType;

    @DataField(pos = 9, precision = 2, length = 12, paddingChar = '0')
    private BigDecimal amount;

    @DataField(pos = 10, length = 3)
    private String currency;

    @DataField(pos = 11, length = 10, pattern = "dd-MM-yyyy")
    private Date orderDate;
```

As of **Camel 2.11** the 'pos' value(s) in a fixed-length record may optionally be defined using ordinal, sequential values instead of precise column numbers.

case 5 : Fixed length record with record-defined field length

Occasionally a fixed-length record may contain a field that define the expected length of another field within the same record. In the following example the length of the instrumentNumber field value is defined by the value of instrumentNumberLen field in the record.

10A9Pauline^M^ISIN10XD12345678BUYShare000002500.45USD01-08-2009

Fixed-delimited

```
@FixedLengthRecord()
public static class Order {

    @DataField(pos = 1, length = 2)
    private int orderNr;

    @DataField(pos = 2, length = 2)
    private String clientNr;

    @DataField(pos = 3, delimiter = "^")
    private String firstName;

    @DataField(pos = 4, delimiter = "^")
    private String lastName;

    @DataField(pos = 5, length = 4)
    private String instrumentCode;

    @DataField(pos = 6, length = 2, align = "R", paddingChar = '0')
    private int instrumentNumberLen;

    @DataField(pos = 7, lengthPos=6)
    private String instrumentNumber;

    @DataField(pos = 8, length = 3)
    private String orderType;

    @DataField(pos = 9, length = 5)
    private String instrumentType;

    @DataField(pos = 10, precision = 2, length = 12, paddingChar = '0')
    private BigDecimal amount;

    @DataField(pos = 11, length = 3)
    private String currency;

    @DataField(pos = 12, length = 10, pattern = "dd-MM-yyyy")
    private Date orderDate;
```

case 6 : Fixed length record with header and footer

Bindy will discover fixed-length header and footer records that are configured as part of the model – provided that the annotated classes exist either in the same package as the primary @FixedLengthRecord class, or within one of the configured scan packages. The following text illustrates two fixed-length records that are bracketed by a header record and footer record.

101-08-2009
10A9 PaulineM ISINXD12345678BUYShare000002500.45USD01-08-2009
10A9 RichN ISINXD12345678BUYShare000002700.45USD01-08-2009
9000000002

Fixed-header-and-footer-main-class

```
@FixedLengthRecord(hasHeader = true, hasFooter = true)
public class Order {

    @DataField(pos = 1, length = 2)
    private int orderNr;

    @DataField(pos = 2, length = 2)
    private String clientNr;

    @DataField(pos = 3, length = 9)
    private String firstName;

    @DataField(pos = 4, length = 5, align = "L")
    private String lastName;

    @DataField(pos = 5, length = 4)
    private String instrumentCode;

    @DataField(pos = 6, length = 10)
    private String instrumentNumber;

    @DataField(pos = 7, length = 3)
    private String orderType;

    @DataField(pos = 8, length = 5)
    private String instrumentType;

    @DataField(pos = 9, precision = 2, length = 12, paddingChar = '0')
    private BigDecimal amount;

    @DataField(pos = 10, length = 3)
    private String currency;

    @DataField(pos = 11, length = 10, pattern = "dd-MM-yyyy")
    private Date orderDate;
...
}


@FixedLengthRecord(isHeader = true)
public  class OrderHeader {
    @DataField(pos = 1, length = 1)
    private int recordType = 1;

    @DataField(pos = 2, length = 10, pattern = "dd-MM-yyyy")
    private Date recordDate;

...
}


@FixedLengthRecord(isFooter = true)
public class OrderFooter {

    @DataField(pos = 1, length = 1)
    private int recordType = 9;

    @DataField(pos = 2, length = 9, align = "R", paddingChar = '0')
    private int numberOfRecordsInTheFile;

...
}
```

case 7 : Skipping content when parsing a fixed length record. (Camel 2.11.1)

It is common to integrate with systems that provide fixed-length records containing more information than needed for the target use case. It is useful in this situation to skip the declaration and parsing of those fields that we do not need. To accomodate this, Bindy will skip forward to the next mapped field within a record if the 'pos' value of the next declared field is beyond the cursor position of the last parsed field. Using absolute 'pos' locations for the fields of interest (instead of ordinal values) causes Bindy to skip content between two fields.

Similarly, it is possible that none of the content beyond some field is of interest. In this case, you can tell Bindy to skip parsing of everything beyond the last mapped field by setting the **ignoreTrailingChars** property on the @FixedLengthRecord declaration.

```
@FixedLengthRecord(ignoreTrailingChars = true)
public static class Order {

        @DataField(pos = 1, length = 2)
        private int orderNr;

        @DataField(pos = 3, length = 2)
        private String clientNr;

    ... any characters that appear beyond the last mapped field will be ignored

}
```

5. Message

The Message annotation is used to identified the class of your model who will contain key value pairs fields. This kind of format is used mainly in Financial Exchange Protocol Messages (FIX). Nevertheless, this annotation can be used for any other format where data are identified by keys. The key pair values are separated each other by a separator which can be a special character like a tab delimitor (unicode representation : \u0009) or a start of heading (unicode representation : \u0001)

"FIX information"

⚠

More information about FIX can be found on this web site : http://www.fixprotocol.org/. To work with FIX messages, the model must contain a Header and Trailer classes linked to the root message class which could be a Order class. This is not mandatory but will be very helpful when you will use camel-bindy in combination with camel-fix which is a Fix gateway based on quickFix project http://www.quickfixj.org/.

Annotation name	Record type	Level
Message	key value pair	Class

Parameter name	type	Info
pairSeparator	string	mandatory - can be '=' or ';' or 'anything'
keyValuePair Separair	string	mandatory - can be '\u0001', '\u0009', '#' or 'anything'
crlf	string	optional - possible values = WINDOWS,UNIX,MAC, or custom; default value = WINDOWS - allow to define the carriage return character to use. If you specify a value other than the three listed before, the value you enter (custom) will be used as the CRLF character(s)
type	string	optional - define the type of message (e.g. FIX, EMX, ...)
version	string	optional - version of the message (e.g. 4.1)
isOrdered	boolean	optional - default value = false - allow to change the order of the fields when FIX message is generated
		This annotation is associated to the message class of the model and must be declared one time.

case 1 : separator = 'u0001'

The separator used to segregate the key value pair fields in a FIX message is the ASCII '01' character or in unicode format '\u0001'. This character must be escaped a second time to avoid a java runtime error. Here is an example :

8=FIX.4.1 9=20 34=1 35=0 49=INVMGR 56=BRKR 1=BE.CHM.001 11=CHM0001-01 22=4 ...

and how to use the annotation

FIX - message

```
@Message(keyValuePairSeparator = "=", pairSeparator = "\u0001", type="FIX", version="4.1")
public class Order {
...
}
```

Look at test cases

The ASCII character like tab, ... cannot be displayed in WIKI page. So, have a look to the test case of camel-bindy to see exactly how the FIX message looks like (src\test\data\fix\fix.txt) and the Order, Trailer, Header classes (src\test\java\org\apache\camel\dataformat\bindy\model\fix\simple\Order.java)

6. KeyValuePairField

The KeyValuePairField annotation defines the property of a key value pair field. Each KeyValuePairField is identified by a tag (= key) and its value associated, a type (string, int, date, ...), optionaly a pattern and if the field is required

Annotation name	Record type	Level
KeyValuePairField	Key Value Pair - FIX	Property

Parameter name	type	Info
tag	int	mandatory - digit number identifying the field in the message - must be unique
pattern	string	optional - default value = "" - will be used to format Decimal, Date, ...
precision	int	optional - digit number - represents the precision to be used when the Decimal number will be formatted/parsed
position	int	optional - must be used when the position of the key/tag in the FIX message must be different
required	boolean	optional - default value = "false"
impliedDecimalSeparator	boolean	**Camel 2.11:** optional - default value = "false" - Indicates if there is a decimal point implied at a specified location

case 1 : tag

This parameter represents the key of the field in the message

FIX message - Tag

```
@Message(keyValuePairSeparator = "=", pairSeparator = "\u0001", type="FIX", version="4.1")
public class Order {

    @Link Header header;

    @Link Trailer trailer;

    @KeyValuePairField(tag = 1) // Client reference
    private String Account;

    @KeyValuePairField(tag = 11) // Order reference
    private String ClOrdId;

    @KeyValuePairField(tag = 22) // Fund ID type (Sedol, ISIN, ...)
    private String IDSource;

    @KeyValuePairField(tag = 48) // Fund code
    private String SecurityId;

    @KeyValuePairField(tag = 54) // Movement type ( 1 = Buy, 2 = sell)
    private String Side;

    @KeyValuePairField(tag = 58) // Free text
    private String Text;

...
}
```

case 2 : Different position in output

If the tags/keys that we will put in the FIX message must be sorted according to a predefine order, then use the attribute 'position' of the annotation @KeyValuePairField

FIX message - Tag - sort

```
@Message(keyValuePairSeparator = "=", pairSeparator = "\\u0001", type = "FIX", version = "4.1", isOrdered =
true)
public class Order {

    @Link Header header;

    @Link Trailer trailer;

    @KeyValuePairField(tag = 1, position = 1) // Client reference
    private String account;

    @KeyValuePairField(tag = 11, position = 3) // Order reference
    private String clOrdId;

...
}
```

7. Section

In FIX message of fixed length records, it is common to have different sections in the representation of the information : header, body and section. The purpose of the annotation @Section is to inform bindy about which class of the model represents the header (= section 1), body (= section 2) and footer (= section 3)

Only one attribute/parameter exists for this annotation.

Annotation name	Record type	Level
Section	FIX	Class

Parameter name	type	Info
number	int	digit number identifying the section position

case 1 : Section

A. Definition of the header section

FIX message - Section - Header

```
@Section(number = 1)
public class Header {

    @KeyValuePairField(tag = 8, position = 1) // Message Header
    private String beginString;

    @KeyValuePairField(tag = 9, position = 2) // Checksum
    private int bodyLength;
...
}
```

B. Definition of the body section

FIX message - Section - Body

```
@Section(number = 2)
@Message(keyValuePairSeparator = "=", pairSeparator = "\\u0001", type = "FIX", version = "4.1", isOrdered =
true)
public class Order {

    @Link Header header;

    @Link Trailer trailer;

    @KeyValuePairField(tag = 1, position = 1) // Client reference
    private String account;

    @KeyValuePairField(tag = 11, position = 3) // Order reference
    private String clOrdId;
```

C. Definition of the footer section

FIX message - Section - Footer

```
@Section(number = 3)
public class Trailer {

    @KeyValuePairField(tag = 10, position = 1)
    // CheckSum
    private int checkSum;

    public int getCheckSum() {
        return checkSum;
    }
```

8. OneToMany

The purpose of the annotation @OneToMany is to allow to work with a List<?> field defined a POJO class or from a record containing repetitive groups.

Restrictions OneToMany

⚠ Be careful, the one to many of bindy does not allow to handle repetitions defined on several levels of the hierarchy

The relation OneToMany ONLY WORKS in the following cases :

- Reading a FIX message containing repetitive groups (= group of tags/keys)
- Generating a CSV with repetitive data

Annotation name	Record type	Level
OneToMany	all	property

Parameter name	type	Info
mappedTo	string	optional - string - class name associated to the type of the List<Type of the Class>

case 1 : Generating CSV with repetitive data

Here is the CSV output that we want :

Claus,Ibsen,Camel in Action 1,2010,35
Claus,Ibsen,Camel in Action 2,2012,35
Claus,Ibsen,Camel in Action 3,2013,35
Claus,Ibsen,Camel in Action 4,2014,35

Remark : the repetitive data concern the title of the book and its publication date while first, last name and age are common

and the classes used to modeling this. The Author class contains a List of Book.

Generate CSV with repetitive data

```
@CsvRecord(separator=",")
public class Author {

        @DataField(pos = 1)
        private String firstName;

        @DataField(pos = 2)
        private String lastName;

        @OneToMany
        private List<Book> books;

        @DataField(pos = 5)
        private String Age;
...


public class Book {

        @DataField(pos = 3)
        private String title;

        @DataField(pos = 4)
        private String year;
```

Very simple isn't it !!!

case 2 : Reading FIX message containing group of tags/keys

Here is the message that we would like to process in our model :

"8=FIX 4.19=2034=135=049=INVMGR56=BRKR"
"1=BE.CHM.00111=CHM0001-0158=this is a camel - bindy test"
"22=448=BE000124567854=1"
"22=548=BE000987654354=2"
"22=648=BE000999999954=3"
"10=220"

tags 22, 48 and 54 are repeated

and the code

Reading FIX message containing group of tags/keys

```
public class Order {

    @Link Header header;

    @Link Trailer trailer;

    @KeyValuePairField(tag = 1) // Client reference
    private String account;

    @KeyValuePairField(tag = 11) // Order reference
    private String clOrdId;

    @KeyValuePairField(tag = 58) // Free text
    private String text;

    @OneToMany(mappedTo = "org.apache.camel.dataformat.bindy.model.fix.complex.onetomany.Security")
    List<Security> securities;
...

public class Security {

    @KeyValuePairField(tag = 22) // Fund ID type (Sedol, ISIN, ...)
    private String idSource;

    @KeyValuePairField(tag = 48) // Fund code
    private String securityCode;

    @KeyValuePairField(tag = 54) // Movement type ( 1 = Buy, 2 = sell)
    private String side;
```

Using the Java DSL

The next step consists in instantiating the DataFormat *bindy* class associated with this record type and providing Java package name(s) as parameter.

For example the following uses the class `BindyCsvDataFormat` (who correspond to the class associated with the CSV record type) which is configured with "com.acme.model"
package name to initialize the model objects configured in this package.

```
// Camel 2.15 or older (configure by package name)
DataFormat bindy = new BindyCsvDataFormat("com.acme.model");


// Camel 2.16 onwards (configure by class name)
DataFormat bindy = new BindyCsvDataFormat(com.acme.model.MyModel.class);
```

Setting locale

Bindy supports configuring the locale on the dataformat, such as

```
// Camel 2.15 or older (configure by package name)
BindyCsvDataFormat bindy = new BindyCsvDataFormat("com.acme.model");
// Camel 2.16 onwards (configure by class name)
BindyCsvDataFormat bindy = new BindyCsvDataFormat(com.acme.model.MyModel.class);

bindy.setLocale("us");
```

Or to use the platform default locale then use "default" as the locale name. Notice this requires Camel 2.14/2.13.3/2.12.5.

```
// Camel 2.15 or older (configure by package name)
BindyCsvDataFormat bindy = new BindyCsvDataFormat("com.acme.model");
// Camel 2.16 onwards (configure by class name)
BindyCsvDataFormat bindy = new BindyCsvDataFormat(com.acme.model.MyModel.class);

bindy.setLocale("default");
```

for older releases you can set it using Java code as shown

```
// Camel 2.15 or older (configure by package name)
BindyCsvDataFormat bindy = new BindyCsvDataFormat("com.acme.model");
// Camel 2.16 onwards (configure by class name)
BindyCsvDataFormat bindy = new BindyCsvDataFormat(com.acme.model.MyModel.class);


bindy.setLocale(Locale.getDefault().getISO3Country());
```

Unmarshaling

```
from("file://inbox")
  .unmarshal(bindy)
  .to("direct:handleOrders");
```

Alternatively, you can use a named reference to a data format which can then be defined in your Registry e.g. your Spring XML file:

```
from("file://inbox")
  .unmarshal("myBindyDataFormat")
  .to("direct:handleOrders");
```

The Camel route will pick-up files in the inbox directory, unmarshall CSV records into a collection of model objects and send the collection to the route referenced by 'handleOrders'.

The collection returned is a **List of Map** objects. Each Map within the list contains the model objects that were marshalled out of each line of the CSV. The reason behind this is that *each line can correspond to more than one object*. This can be confusing when you simply expect one object to be returned per line.

Each object can be retrieve using its class name.

```
    List<Map<String, Object>> unmarshaledModels = (List<Map<String, Object>>) exchange.getIn().getBody();

    int modelCount = 0;
    for (Map<String, Object> model : unmarshaledModels) {
          for (String className : model.keySet()) {
             Object obj = model.get(className);
             LOG.info("Count : " + modelCount + ", " + obj.toString());
          }
         modelCount++;
    }

    LOG.info("Total CSV records received by the csv bean : " + modelCount);
```

Assuming that you want to extract a single Order object from this map for processing in a route, you could use a combination of a Splitter and a Processor as per the following:

```
from("file://inbox")
    .unmarshal(bindy)
    .split(body())
        .process(new Processor() {
            public void process(Exchange exchange) throws Exception {
                Message in = exchange.getIn();
                Map<String, Object> modelMap = (Map<String, Object>) in.getBody();
                in.setBody(modelMap.get(Order.class.getCanonicalName()));
            }
        })
        .to("direct:handleSingleOrder")
    .end();
```

Take care of the fact that Bindy uses CHARSET_NAME property or the CHARSET_NAME header as define in the Exchange interface to do a characterset conversion of the inputstream received for unmarshalling. In some producers (e.g. file-endpoint) you can define a characterset. The characterset conversion can already been done by this producer. Sometimes you need to remove this property or header from the exchange before sending it to the unmarshal. If you don't remove it the conversion might be done twice which might lead to unwanted results.

```
from("file://inbox?charset=Cp922")
  .removeProperty(Exchange.CHARSET_NAME)
  .unmarshal("myBindyDataFormat")
  .to("direct:handleOrders");
```

Marshaling

To generate CSV records from a collection of model objects, you create the following route :

```
from("direct:handleOrders")
   .marshal(bindy)
   .to("file://outbox")
```

Using Spring XML

This is really easy to use Spring as your favorite DSL language to declare the routes to be used for camel-bindy. The following example shows two routes where the first will pick-up records from files, unmarshal the content and bind it to their model. The result is then send to a pojo (doing nothing special) and place them into a queue.

The second route will extract the pojos from the queue and marshal the content to generate a file containing the csv record. The example above is for using Camel 2.16 onwards.

spring dsl

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://camel.apache.org/schema/spring
       http://camel.apache.org/schema/spring/camel-spring.xsd">

        <!-- Queuing engine - ActiveMq - work locally in mode virtual memory -->
        <bean id="activemq" class="org.apache.activemq.camel.component.ActiveMQComponent">
                <property name="brokerURL" value="vm://localhost:61616"/>
        </bean>


        <camelContext xmlns="http://camel.apache.org/schema/spring">


        <dataFormats>
          <bindy id="bindyDataformat" type="Csv" classType="org.apache.camel.bindy.model.Order"/>
        </dataFormats>

                <route>
                        <from uri="file://src/data/csv/?noop=true" />
                        <unmarshal ref="bindyDataformat" />
                        <to uri="bean:csv" />
                        <to uri="activemq:queue:in" />
                </route>

                <route>
                        <from uri="activemq:queue:in" />
                        <marshal ref="bindyDataformat" />
                        <to uri="file://src/data/csv/out/" />
                </route>
        </camelContext>
</beans>
```

Be careful

Please verify that your model classes implements serializable otherwise the queue manager will raise an error

Dependencies

To use Bindy in your camel routes you need to add the a dependency on **camel-bindy** which implements this data format.

If you use maven you could just add the following to your pom.xml, substituting the version number for the latest & greatest release (see the download page for the latest versions).

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-bindy</artifactId>
  <version>x.x.x</version>
</dependency>
```