Cocoon Forms: Datatypes (2.1 legacy document)

Table of contents

Table of contents

Warning:
This document was copied as is from the Cocoon 2.1 documentation, but has not yet been fully reviewed or moved to its new home.

1. Context

Datatypes are used by certain widgets, more specifically the field and multivaluefield widgets, to allow them to edit different types of data (e.g. strings, numbers, dates). For a general introduction of the relationship between the widget and the datatype, see the documentation of the field widget.

2. General information

In its most basic form a datatype is declared as follows:

```
<fd:datatype base="...">
```

The **base** attribute refers to one of the built-in datatypes such as string or long.

2.1. Convertors

A datatype needs a convertor. The purpose of a convertor is to convert between string and object representations of values. There is always a default convertor, but you change or configure that using the fd:convertor element. Here's an example for dates:

```
<fd:datatype base="date"> <fd:convertor type="formatting"> <fd:patterns>
<fd:pattern>dd/MM/yyyy</fd:pattern> </fd:patterns> </fd:convertor> </fd:datatype>
```

The **type** attribute on the fd:convertor element is optional, if not specified the default one will be used (configured in the cocoon.xconf). Any further content of the fd:convertor element is specific to the convertor implementation and will be documented in a seperate section.

2.2. Selection lists (default implementation)

Widgets that have a datatype can also have a selection list. Since selection lists are associated with datatypes, we discuss them here. The selection list can be defined inline or read from an external source. Example of inline declaration:

```
<fd:datatype base="long"/> <fd:selection-list> <fd:item value="1"/> <fd:item value="2"/>
<fd:item value="3"> <fd:label>three</fd:label> </fd:item> <fd:item value="4"/> <fd:item
value="5"/> </fd:selection-list>
```

Each item in the selection-list can have a value (specified in the value attribute) and optionally a label (specified in the fd:label element). If no label is specified, the value is used as label. The fd:label element can contain mixed content.

To set a default selection, just set the value of the widget containing the selection list.

Example of getting a selection list from an external source:

```
<fd:datatype base="string"/> <fd:selection-list src="cocoon:/mychoices.xml"/>
```

All Cocoon-supported protocols can be used. The format of the XML produced by the source should be the same as in case of inline specification of the selection list, thus the root element should be a fd:selection-list element.

By default, the selection list will be retrieved form the source once, and then become part of the form definition, just like when you would have defined it inline. This has the consequence that if the XML

produced by the source changes, you won't see the selection list changed. If you'd like CForms to retrieve the content of the selection list each time it needs it, add an attribute called "dynamic" with value "true", for example:

```
<fd:datatype base="string"/> <fd:selection-list src="cocoon:/mychoices.xml" dynamic="true"/>
```

If the datatype is different from string, CForms will need to convert the string values that appear in the selection list to their object equivalent. This conversion is normally done using the same convertor as the datatype in which the selection list appears, but you can also specify a different one. Here's an example for a date selection list:

```
<fd:datatype base="date"/> <fd:selection-list> <fd:convertor type="formatting"> <fd:patterns> <fd:pattern>yyyyMMdd</fd:pattern> </fd:patterns> </fd:convertor> <fd:item value="13020711"/> <fd:item value="19120623"/> <fd:item value="19690721"/> <fd:item value="19700506"/> <fd:item value="19781014"/> <fd:item value="20010911"/> </fd:selection-list>
```

If there is a fd:convertor element, it should always be the first child element of the fd:selection-list element. This works of course also for selection lists retrieved from external sources.

Selection list implementations are pluggable. Everything said until now applies to the default selection list implementation. An alternative implementation can be specified by using a **type** attribute on the fd:selection-list element. The sections below describe the alternative implementations currently available.

2.3. Selection lists: flow-jxpath implementation

See the javadoc of the FlowJXPathSelectionListBuilder class for now.

Example:

In flowscript:

```
var data = new Object(); data.cityList = new Array(2); data.cityList[0] = {value:"AL", label:"Alabama"}; data.cityList[1] = {value:"AK", label:"Alaska"}; form.showForm("flow/myform.form", data);
```

and the corresponding selection list definition:

```
<fd:selection-list type="flow-jxpath" list-path="cityList" value-path="value" label-path="label" />
```

Hint: the label can be any kind of object, its toString() method will be called to get the string to be displayed. In case the object supplied as label implements the XMLizable interface, its toSAX method will be called instead. One practical application of this is using i18n labels:

```
importClass (Packages.org.apache.cocoon.forms.util.I18nMessage); ... mylist[0] = {value: "x", label: new I18nMessage("myI18nKey") };
```

2.4. Selection lists: enum implementation

This type of selection list outputs a list of items corresponding to the possible instances of an enumerated type (see below).

Example:

```
<fd:selection-list type="enum" class="com.example.Sex"/>
```

outputs:

```
<fi:selection-list> <fi:item value=""/> <fi:item value="com.example.Sex.MALE"> <fi:label>
```

<i18n:text>com.example.Sex.MALE</i18n:text> </fi:label> </fi:item> <fi:item value="com.example.Sex.FEMALE"> <fi:label> <i18n:text>com.example.Sex.FEMALE</i18n:text> </fi:label> </fi:item> </fi:selection-list>

If you don't want an initial null value, add a **nullable="false"** attribute to the **fd:selection-list** element. This applies only to **enum** type selection lists.

3. Available datatypes

CForms datatype	Java class	Convertors
string	java.lang.String	Strings obviously don't support any convertors, since there's no purpose in converting a string to a string.
decimal	java.math.BigDecimal	formatting (decimal), plain
integer	java.lang.Integer	similar as decimal datatype
long	java.lang.Long	similar as decimal datatype
double	java.lang.Double	similar as decimal datatype
date	java.util.Date	formatting (date), millis
enum	Enumerated type	enum

3.1. Enumerated datatype

The **enum** datatype is meant to be used with types implementing Joshua Bloch's typesafe enum pattern. The following is a possible implementation:

package com.example; public class Sex { public static final Sex MALE = new Sex("M"); public static final Sex FEMALE = new Sex("F"); private String code; private Sex(String code) { this.code = code; } }

The following snippet shows the usage of this type:

<fd:field id="sex"> <fd:label>Sex</fd:label> <fd:datatype base="enum"> <fd:convertor type="enum"> <fd:enum>com.example.Sex</fd:enum> </fd:convertor> </fd:datatype> <fd:selection-list type="enum" class="com.example.Sex"/> </fd:field>

If your enumerated type does not provide a toString() method, the enum convertor will use the fully qualified class name, followed by the name of the public static final field referring to each instance, i.e. "com.example.Sex.MALE", "com.example.Sex.FEMALE" and so on.

If you provide a toString() method which returns something different, you should also provide a fromString(String, Locale) method to convert those strings back to instances.

The enum datatype is typically used together with the **enum** selection list type.

4. Available convertors

4.1. formatting (decimal)

This convertor uses the java.text.DecimalFormat class (or com.ibm.icu.text.DecimalFormat class if it is present in the classpath). This means it can perform locale-dependent, pattern-based formatting of

numbers.

Configuration pseudo-schema:

<fd:convertor type="formatting" variant="integer|number|currency|percent" ? > <fd:patterns> <fd:pattern>....</fd:pattern> ? <fd:pattern locale="lang-COUNTRY">....</fd:pattern> * </fd:patterns> ? </fd:convertor>

The variant attribute and patterns element are optional. By default, the "number" variant is used (or for longs: the "integer" variant).

You can supply either a locale-independent formatting pattern or locale-dependent formatting patterns. See the javadoc of the DecimalFormat class for the supported pattern syntax. CForms will always use the pattern that is most specific for the current locale.

4.2. plain

This convertor is not locale-dependent. It shows the full precision of the number and uses dot as the decimal separator.

4.3. date convertors

The date datatype can be used both for dates as times. The date datatype supports the following convertors:

4.3.1. formatting (date)

This convertor uses the java.text.SimpleDateFormat class (or com.ibm.icu.text.SimpleDateFormat class if it is present in the classpath). This means it can perform locale-dependent, pattern-based formatting of dates.

Configuration pseudo-schema:

<fd:convertor type="formatting" variant="date|time|datetime" ? style="short|medium|long|full" ?> <fd:patterns> <fd:pattern>....</fd:pattern> ? <fd:pattern locale="lang-COUNTRY">....</fd:pattern> * </fd:patterns> ? </fd:convertor>

Usually you will use either the variant and style attributes or the pattern(s).

For example, the following convertor configuration:

<fd:convertor type="formatting" variant="date" style="short">

Will give the following for July 15, 2003: 7/15/03. Using style medium it gives "Jul 15, 2003", style long gives "July 15, 2003", and style full gives "Tuesday, July 15, 2003". These result are locale-dependent of course.

Here's an example of using a formatting pattern:

<fd:convertor type="formatting"> <fd:patterns> <fd:pattern>dd/MM/yyyy</fd:pattern> </fd:patterns> </fd:convertor>

Using the same date, this will now give "15/07/2003".

It is also possible to use different patterns for different locales by using multiple fd:pattern elements with "locale" attributes, for example:

<fd:convertor type="formatting"> <fd:patterns> <fd:pattern>MM/dd/yyyy</fd:pattern> <fd:pattern locale="nl-BE">dd/MM/yyyy</fd:pattern> <fd:pattern

locale="fr">dd-MM-yyyy</fd:pattern> </fd:patterns> </fd:convertor>

In this case, if the locale is "nl-BE", the second pattern will be used; if the locale is "en", the first pattern will be used; and if the locale is "fr-BE" the third pattern will be used (because when fr-BE is not found, it will first search for "fr" before using the locale-indepent pattern).

4.3.2. millis

The millis convertor for dates uses the number of milliseconds since January 1, 1970, 00:00:00 GMT as string representation. This will likely not be used to present dates to the user, but may be useful in selection lists retrieved from external sources.

1. Comments

add your comments