Calyxo Forms

Table of contents

1 Introduction....4
2 Forms Concepts....4
2.1 Configuration....5
2.2 Validators....7
2.3 Forms....9
2.4 Fields....10
2.5 Inputs....12
2.6 Messages....13
2.7 Assertions....14
2.8 Actions....15
2.9 Tag Library....16
2.10 Selections....24
2.11 I18n....28
3 Reference....29
3.1 Configuration....29
3.1.1 The <calyxo-forms-config> Element....30
3.1.2 The <validators> Element....30
3.1.3 The <matcher> Element....31
3.1.4 The <converter> Element....31
3.1.5 The <checker> Element....32
3.1.6 The <property> Element....32
3.1.7 The <message> Element....33
3.1.8 The <arg> Element....34
3.1.9 The <forms> Element....35
3.1.10 The <form> Element....36
3.1.11 The <field> Element....36
3.1.12 The <match> Element....37

3.1.13 The <convert> Element....37
3.1.14 The <check> Element....38
3.1.15 The <input> Element....38
3.1.16 The <assert> Element....39
3.2 Accessors....39
3.2.1 The forms.data accessor....40
3.2.2 The forms.list accessor....40
3.3 Tag Library....40
3.3.1 The <debug> Tag....42
3.3.2 The <form> Tag....42
3.3.3 The <text> Tag....43
3.3.4 The <password> Tag....44
3.3.5 The <hidden> Tag....46
3.3.6 The <textarea> Tag....46
3.3.7 The <checkbox> Tag....47
3.3.8 The <radio> Tag....48
3.3.9 The <group> Tag....49
3.3.10 The <checkboxitem> Tag....50
3.3.11 The <radioitem> Tag....51
3.3.12 The <select> Tag....52
3.3.13 The <option> Tag....53
3.3.14 The <listoption> Tag....54
3.3.15 The <listoptions> Tag....54
3.4 Predefined Validators....55
3.4.1 Predefined Matchers....57
3.4.2 Predefined Converters....57
3.4.3 Predefined Checkers....59
4 Extension Points....60
4.1 Adding your own validators....61
5 Integration....63
5.1 Forms Plugin for Calyxo....64
5.2 Forms Plugin for Struts....66
6 Project....66
6.1 History of Changes....66

6.2 Todo List.....69

1. Introduction

The *Calyxo Forms* component allows the definition of forms, along with all their validation steps and lets you map inputs to properties of a data bean. Complex validations, dependencies between inputs and mapping an input to a set of properties are supported.

Flexibility

Calyxo Forms offers you a large amount of flexibility for its main task: input validation.

It comes packed with lots of so called *validators*, which handle all aspects of single field validation, for example regular expression matching or conversion into date values. You may also define your own validators, if you have special requirements.

Furthermore it lets you define validation rules for two or more input fields of the same form, they are called *assertions*. Here you can use a set of standard operators and functions to express your validation conditions. And again, if you like to, you can define your own set of functions to be used in your assertions.

Reusability

Calyxo was designed with reusability aspects in mind. As far as validation is concerned, this means, that you will define your validators, functions or messages at one place and will use them where ever you like to.

Portability

You may use *Calyxo Forms* with *Calyxo Control* or with Struts. The information provided on the following pages apply to either environment.

Recommended Reading

If you are new to the *Calyxo Forms* component and like to get an idea what it's all about, you might look into the Concepts section. When you are starting to work with *Calyxo Forms*, the Reference section will hopefully provide you with all required details. Finally you may want to write your own extensions, making *Calyxo Forms* even more powerful, in this case you should find the desired information in the Extension Points section.

2. Forms Concepts

The central notion of the *Calyxo Forms* component is that of a *validator*, which is used for the validation of a single input field. Additionally it allows the definition of validation rules concerning two or more input fields of a form, the so called *assertions*. Below we'll have a glance at these ideas, which will be explained in more detail on the following pages.

Validators

Calyxo Forms comes with lots of so called *validators*, which handle the standard tasks evolving during input validation: for example, they match regular expressions, they convert input strings into standard data types like numbers or dates, and they allow you to check the resulting property values to fulfill your requirements.

All of the *Calyxo Forms* validators are ready to use, but if you feel the need for additional validators, *Calyxo Forms* offers an extension mechanism, which allows the definition of custom validators.

Assertions

Whereas the validators are aimed at single field validation, *Calyxo Forms* does offer you a powerful tool for the additional validation of dependencies between two or more input fields of the same input form: the so called *assertions*.

An assertion allows you to write your validation condition as a boolean expression, which may contain variables referring to input strings or already converted property values. The expressions may contain many standard functions, but again: if you are missing anything, *Calyxo* allows you to enhance the expression's features by adding your own set of functions.

Error messages

As usually all *Calyxo Forms* messages are kept in message bundles and referred by a message key. They can be associated with a validator or an assertion. In addition to returning an error message describing the invalid input, *Calyxo Forms* allows you easily to mark the corresponding input fields optically (typically by turning the background into red).

Internationalization

I18n issues turn out to be relevant for input validation at three different places:

- The input data has to be parsed in a locale specific manner, since dates or numbers have different string representations in different countries.
- The same reason causes the formatting of property values to be locale dependent.
- The error messages, which are caused by invalid input data, have to be localized.

Not only does the *Calyxo Forms* component provide support to handle all these issues easily. It also offers the possibility of defining locale dependent form definitions, which allow localized assertions as well.

2.1. Configuration

As usual, the *Calyxo Forms* component is configured per module in one or more XML configuration files.

Namespace

Each *Calyxo* component uses its own separate namespace for configuration elements. The elements described in the following sections are defined within namespace `http://calyxo.odysseus.de/xml/ns/forms`.

Schema/DTD Validation

When loading configuration files, *Calyxo* forces the XML parser to validate documents against an XML Schema definition (XSD) or a Document Type Declaration (DTD). *Calyxo* prefers XML Schema validation, so, if your parser supports it, it is used. Otherwise, DTD validation is used. In this case, configuration files must declare the *Calyxo Forms* document type:

```
<!DOCTYPE calyxo-forms-config
  PUBLIC "-//Odysseus Software GmbH//DTD Calyxo Forms 0.9//EN"
  "calyxo-forms-config.dtd">
```

Copies of the DTD and XSD are located at `CALYXO_HOME/calyxo-forms/conf/share/calyxo-forms-config.*`.

Document structure

The root element is `<calyxo-forms-config>`. As common to all of *Calyxo*'s configuration files, the root element specifies the `xmlns` and `version` attributes and may optionally contain `<base:import>` elements, followed by `<base:functions>`, `<base:set>` and `<base:use>` elements. See the *Calyxo Base* configuration section for a description of these elements. Using them requires the declaration of namespace `http://calyxo.odysseus.de/xml/ns/base` for prefix `base` (as in the document template below).

Following these common elements, the root contains an optional `<validators>` element and zero or more `<forms>` elements. The `<validators>` element contains one or more validator definitions. A `<forms>` element can be associated with a locale by specifying the `language`, `country` and `variant` attributes. This advanced feature is covered in the i18n section.

For now, let's consider the usual case, where we have exactly one `<forms>` section, without specifying a locale. Thus, our configuration document looks like this:

```
<calyxo-forms-config version="0.9"
  xmlns="http://calyxo.odysseus.de/xml/ns/forms"
  xmlns:base="http://calyxo.odysseus.de/xml/ns/base">

  <!-- base:import elements can go here -->
  <!-- base:functions, base:set and base:use elements
       can go here -->

  <!-- a validators element can go here -->
```

```
  <forms>

    <!-- form definitions go here -->

  </forms>

</calyxo-forms-config>
```

A <forms> element contains one or more form definitions.

2.2. Validators

The validation of a single input field is done by a sequence of so called **validators**. Every validator does process the current input value in some way and passes the result to the next validator in the sequence (or, if it fails to process, the validation of this input field is cancelled). The sequence of validators is divided into three parts (matching, converting and checking), each of them containing a specific type of validators (matchers, converters and checkers, respectively). Let's have a closer look at the three phases of validation:

1. **Matching:** The first phase of input validation is done by a **sequence** of matchers. A **matcher** takes the current input value (a string) and accepts it or not (according to his predefined rules). If it does accept the input string, it returns a result string, which is then taken as the current input field value and passed to the next validator (i.e., another matcher or a converter). Typical examples for matchers are the `TrimMatcher`, which removes leading and trailing spaces, and the `RegExpMatcher`, which matches the input string against a given regular expression (both can be found in the `de.odysseus.calyxo.forms.match` package, where the predefined *Calyxo Forms* matchers are implemented).
2. **Converting:** The second phase of input validation is done by a **single** converter. A **converter** takes the current input value (a string) and tries to convert it to a value from its corresponding domain (i.e., booleans, numbers, dates, etc.). If the input string is a valid string representation of such a value, this value is taken as the current input field value and passed to the next validator (i.e., a checker). Typical examples for converters are the `BooleanConverter`, which converts the strings `"true"` and `"false"` to the corresponding values of `java.lang.Boolean`, and the `LongConverter`, which converts to `java.lang.Long` instances (both can be found in the `de.odysseus.calyxo.forms.convert` package, where the predefined *Calyxo Forms* converters are implemented).
3. **Checking:** The third phase of input validation is done by a **sequence** of checkers. A **checker** takes the current input value and accepts it or not (according to his predefined rules). A checker does not change the current input value. If the input is accepted the validation continues with the next validator (i.e., a checker), if there is one. Typical examples for checkers are the `LengthChecker`, which assures that the length of a string is in a given interval, and the `RangeChecker`, which does a similar job for a numerical value. (both can be found in the `de.odysseus.calyxo.forms.check` package, where the predefined *Calyxo Forms* checkers are implemented).

Implementation, declaration and usage

In general, to use a validator for the validation of an input field the following three steps have to be executed:

1. **Implementation:** Write a Java class implementing one of the three validator interfaces defined in the `de.odysseus.calyxo.forms` package: `Matcher`, `Converter` or `Checker`. Chances are good, that you are not required to do so too soon, since *Calyxo Forms* offers a whole bunch of validators covering standard tasks. If you are interested in writing custom validators, have a look into the Validators section of the extension points documentation.
2. **Declaration:** Add a `<matcher>`, `<converter>` or `<checker>` element to the `<validators>` section of your configuration file, which refers to your class and defines a name, properties and a message for your new validator. This will be explained in more detail below.
3. **Usage:** Add a `<match>`, `<convert>` or `<check>` element, which refers to the name of the validator defined in step 2., to every `<field>` element of your configuration file's `<forms>` section, where you like to use your validator. This is the only step you need to care about, when you're satisfied with *Calyxo*'s built-in validators. It will be further illustrated in the Fields section.

Declaring a custom validator

As mentioned above, you need to declare your custom validators in the `<validators>` section of your forms configuration file. Even if you did not implement any validators as Java classes, you might find it useful, to declare a validator using one of *Calyxo*'s built-in validator classes. All of them are already declared in the file `calyxo-validators.xml`, a copy of which can be found in the directory `CALYXO_HOME/calyxo-forms/conf/share/`. When using them in your form validation, you can change their properties and/or message whenever you need. But if you intend to do so in the same way at many different input fields, it might be easier to declare a custom validator.

Let's look at an example defining a customized version of the `LengthChecker`, which is used to check the length of an input string:

```
<validators>
  <checker id="length8"
    class="de.odysseus.calyxo.forms.check.LengthChecker">

    <property name="min" value="8"/>
    <property name="max" value="8"/>

    <message key="error.check.length8" bundle="foo.msg">
      <arg name="field"/>
    </message>

  </checker>
</validators>
```

Now the checker with name `"length8"` is ready to be used for field validation. It will check the

given input string to consist of exactly eight characters (since both of it's properties, i.e. `min` and `max` have been set to 8) and will generate an appropriate error message if the test fails. The message has to be configured in the file `msg.properties` in package `foo`, for example with the entry `"error.check.length8=Field '{0}' must have a length of 8"`. The single argument of the message has been named `"field"` by placing `<arg name=…>` in the body of the `message` element. This increases the readability of the field validations, especially for messages with more than one argument.

Let's have a glance at using your new validator, though this is not our topic in this section. To check the input for property `bar` to be exactly of length eight (and generate the message `"Field 'Bar' must have a length of 8"` if it fails) you simply add the following lines to your `<form>` definition:

```
<field property="bar">

  <check name="length8">
    <message>
      <arg name="field" value="Bar"/>
    </message>
  </check>

</field>
```

2.3. Forms

A *Validation Form* definition contains all the information required to validate the input data contained in a form. On the one hand these are single-field validation rules (expressed via validators), on the other hand we have multiple-field validation rules (expressed via assertions).

These form definitions are contained in the `<forms>` section of a *Calyxo Forms* configuration file. (Actually, there may be more than one `<forms>` section for different locales, but for the sake of simplicity we assume a single `<forms>` section here.)

A form definition consists of two parts (both of which may be empty). It starts with the single-field validation rules defined in `<field>` and `<input>` elements, followed by the multiple-field validation rules defined in `<assert>` elements. Let's look at an example:

```
<form name="orForm">

  <field property="date1">
    <convert name="date">
      <message>
        <arg name="field" value="date1"/>
      </message>
    </convert>
  </field>

  <field property="date2">
```

```
    <convert name="date">
      <message>
        <arg name="field" value="date2"/>
      </message>
    </convert>
  </field>

  <assert test="not empty input.date1 or not empty input.date2">
    <message bundle="messages" key="error.or">
      <arg value="date1"/>
      <arg value="date2"/>
    </message>
  </assert>

</form>
```

Here we have two input properties, `date1` and `date2`, which are converted from input strings into date values using the date converter. If one of the inputs can not be parsed as a date, a suitable error message specifying the input field is generated. If both inputs are valid (i.e., none of their validators failed), the assertion assures that not both of them are empty (empty inputs do not cause the date converter to fail, it simply converts them into a `null` date). If the assertion fails, it generates a custom error message, which might be defined as `"error.or=At least one of '{0}' and '{1}' must be entered"`. Only when the input passes all validators and assertions, the converted values are stored in the form properties and the action, which caused the validation, is executed.

2.4. Fields

The `<field>` element is used to specify the validation rules for a single input and to map the validation result to a form data property. After a form has successfully been validated, valid field values can populate their corresponding form data properties.

`<field>` elements may appear either as a children of a `<form>` element or inside `<input>` elements.

The `<field>` element requires the `property` attribute, giving the form data property name, to which this field maps. If the `<field>` is a direct `<form>` child, the `property` also gives th HTTP parameter name.

Actually, specifying a `<field>` element as a `<form>` child is just an abbreviated syntax for the common case, where a scalar input parameter is mapped to a single property with the same name. In other words, the form definitions

```
<form name="foo">
  <field property="bar">
    ...
  </field>
</form>
```

and

```
<form name="foo">
  <input name="bar">
    <field property="bar">
      ...
    </field>
  </input>
</form>
```

are equivalent.

The `<field>` element may contain

1. a sequence of `<match>` elements, configuring the matchers to be used in the matching phase,
2. a `<convert>` element, configuring the converter to be used in the converting phase,
3. a sequence of `<check>` elements, configuring the checkers to be used in the checking phase,
4. a `<message>` element, configuring a message to be generated when validation fails.

Field Validations

Field validations make use of validators, that have been declared inside the `<validators>` element (or of the predefined validators). Validations are applied in the order the corresponding elements appear inside the `<field>` element. Per definition, a field is valid, if all of its validations succeed.

All of the `<match>`, `<convert>` and `<check>` elements require the `name` attribute to reference a validator. They may contain `<property>` elements and a `<message>` element to customize the validator's properties and message.

Nested `<property>` elements can be used to set the validator properties, that it defines. A nested `<message>` element can be given to customize the validator's message.

For example, if a decimal input value shall be checked to be less than `10`, the following field definition should be used:

```
<field property="bar">
  <convert name="decimal">
    <property name="maximumFractionDigits" value="2"/>
    <message>
      <arg name="field" value="Bar"/>
    </message>
  </convert>
  <check name="less">
    <property name="max" value="10.0"/>
    <message>
      <arg name="field" value="Bar"/>
```

```
    </message>
  </check>
</field>
```

We can "factor out" the common message argument named `field` and rewrite the above as:

```
<field property="bar">
  <convert name="decimal">
    <property name="maximumFractionDigits" value="2"/>
  </convert>
  <check name="less">
    <property name="max" value="10.0"/>
  </check>
  <message>
    <arg name="field" value="Bar"/>
  </message>
</field>
```

To make sure that a required date input is no Sunday, we write:

```
<field property="departure">
  <match name="notEmpty"/>
  <convert name="date"/>
  <check name="el">
    <property name="expression" value="property.day ne 0"/>
    <message bundle="foo.messages" key="error.check.noSunday">
      <arg value="Departure Date"/>
    </message>
  </check>
  <message>
    <arg name="field" value="Departure Date"/>
  </message>
</field>
```

By the way, the last example illustrated the use of an EL expression checker, which checks its expression to evaluate to `true` (the value to be checked is referred by the `property` variable, as usually the appending `.day` causes the `getDay()` method of our `Date` object to be called). The EL checker does usually require a custom message (since the expressions are not user-readable), so we add an entry like `"error.check.noSunday=Field '{0}' must not contain a Sunday date"` in our `messages.properties` file.

2.5. Inputs

An `<input>` element corresponds to a HTTP form input parameter, given by the mandatory `name` attribute. Optional attributes are

- `array` - a `boolean`, specifying if this input is a scalar or array input. The default value is `false`.
- `ignore` - a boolean expression (without leading `'${'` and trailing `'}'`); if specified, the

expression will be evaluated before validating the input. If it evaluates to `true`, validation will be skipped for that input. If a form is valid or not does not depend on ignored inputs.
- `relax` - a boolean expression (without leading `'${'` and trailing `'}'`); if specified, the expression will be evaluated when validating the input has failed. If it evaluates to `true`, no error message will be generated. However, the form is still invalid.

An `<input>` element contains one or more `<field>` elements. Each field maps to a different property. Per definition, an input is valid, if one of its fields is valid. The first valid field will get its value mapped to its form data property. All other fields will map `null` to their form data properties.

When validating an input fails, a message will be generated (except if it its `relaxed` expression evaluates to `true`). The default is to take the message from the last nested field (the last field for which validation failed). Explicitly specifying a message on input level overrides this behavior.

2.6. Messages

Messages are defined and configured using the `<message>` element.

A `<message>` element may contain `<arg>` children to specify message arguments. An argument may specify its value in one of the following ways:

- the `value` attribute directly contains the message argument. Since it is dynamic, it may be given as an expression.
- the `bundle` and `key` attributes specify a localized resource containing the argument value.
- the `property` attribute specifies a validator property, containing the argument value (only allowed for messages inside validators and field validations).

Validators, defined in the `<validators>` section, are used by validations, defined in a `<form>` element. Correspondingly, a message, that has been defined for a validator, may be customized by messages, defined in validations, that use this validator. For example, if a message contains the field name as an argument, this is usually not known by the validator. It is, however, known inside the form, where we use the validator in a field validation.

To enable customization of validator message arguments, `<arg>` elements in validator messages *may* have a `name` attribute, meaning "the value for this argument may be specified or overridden by validation messages". On the other hand, validation message arguments *must* have the `name` attribute set to reference an argument in the base validator message.

Additionally, we allow `<message>` elements within fields. A field message is used to "factor out" message arguments which are shared across several field validation messages inside that field. For example, most predefined validators declare an argument named `field` to denote the field name. Though rarely used, field messages can also completely override all validator messages for that field by specifying a `key`.

We therefore distinguish three semantically and syntactically different ways to use the `<message>` element.

1. A *simple message definition* may appear in `<input>`, `<assert>`, `<match>`, `<convert>` and

<check> elements. In this case, the message and its arguments are always fully described. Both, the bundle and key attributes, are required. Nested <arg> elements do not have name attributes.

2. A *base message definition* may appear inside validator elements <matcher>, <converter> and <checker> as well as in <field> elements. Parts of the message details may be left open: the key attribute is required, but the bundle attribute may be omitted. Nested <arg> elements may specify a name attribute. If they do so, they may also leave their values unspecified (abstract argument). A base message inside a <field> element shadows (completely overrides) a base message in a validator.
3. A *message configuration* may appear inside validation elements <match>, <convert> and <check> as well as in <field> elements. Here, parts of a base message definition are added or overridden. The key attribute is forbidden, but the bundle attribute may be specified. Nested <arg> elements must specify a name attribute. named arguments from their base definition.

2.7. Assertions

Assertions are used to validate complex conditions among multiple form fields or input parameters. The <assert> element specifies a boolean EL expression in its mandatory test attribute.

Assertions are the last step in form validation. Assert expressions are evaluated in a variable context, which allows the following implicit objects:

- requestScope, sessionScope, moduleScope, applicationScope - request, session, module, application scope maps.
- param - HTTP parameter map.
- moduleContext - module context instance.
- input - Form input map, associating form input names with the unvalidated form input strings.
- property - Form data map, associating form data property names with the validated form data values.

During expression evaluation, all inputs referenced by the implicit objects input and property are collected. If the expression evaluates to false, the following rules apply:

- If the result depends on a reference to an invalid input, it will be ignored.
- Otherwise, the assertion has failed and the referenced inputs are marked.

The nested <message> element gives the message to be generated if the assertion fails.

Example

The following example shows a form definition with two input fields (account number and creditcard number). The assertion at the end assures that exactly one of the fields has been filled with valid data (i.e. a long value to keep it simple).

```
<form name="bankData">
```

```
  <field property="account">
    <convert name="long">
      <property name="groupingUsed" value="false"/>
      <message>
        <arg name="field" bundle="foo.msg" key="label.account"/>
      </message>
    </convert>
  </field>
  <field property="creditcard">
    <convert name="long">
      <property name="groupingUsed" value="false"/>
      <message>
        <arg name="field" bundle="foo.msg" key="label.creditcard"/>
      </message>
    </convert>
  </field>
  <assert test="empty property.account != empty property.creditcard">
    <message bundle="foo.msg" key="error.oneOf">
      <arg bundle="foo.msg" key="label.account"/>
      <arg bundle="foo.msg" key="label.creditcard"/>
    </message>
  </assert>
</form>
```

The required entries in your `msg.properties` file might look like these:

```
error.oneOf=Either field '{0}' or field '{1}' has to be filled (not both)

label.account=Account Number
label.creditcard=Creditcard Number
```

2.8. Actions

After the input values passed all the pre-configured single-field and multiple-field validation rules, the corresponding action is executed and may perform final checks on the input data. If they are successful, the properties are committed: their values are stored in the corresponding form data (assuming you didn't turn on the auto-commit feature).

The module's `de.odysseus.calyxo.forms.FormsSupport` instance serves as a central point for actions to interact with *Calyxo Forms*. Everything starts by getting a reference to the `FormsSupport` instance:

```
FormsSupport support = FormsSupport.getInstance(request);
```

The `FormsSupport` instance offers you the following methods:

`public FormProperties getFormProperties(HttpServletRequest request, String action)`
Returns the form properties (of type `de.odysseus.calyxo.forms.FormProperties`) for the

specified request and action.

`getFormData(HttpServletRequest request, String action, boolean create)`
Returns the form data (of type `de.odysseus.calyxo.forms.FormData`) for the specified request and action. (If there is none and the `create` flag is set to `true`, an appropriate one is created.)

`removeFormData(HttpServletRequest request, String action)`
Removes the form data for the specified request and action.

`getFormResult(HttpServletRequest request, String action)`
Returns the form result (of type `de.odysseus.calyxo.forms.FormResult`) for the specified request and action.

Form Properties

An instance of `de.odysseus.calyxo.forms.FormProperties` gives access to the validated (and converted) input values via its `getProperty(String name)` method:

```
FormProperties properties = support.getFormProperties(request, "/foo");
Date arrival = (Date)properties.getProperty("arrival");
```

Depending on your environment, there may be action base classes, which do this step for you and pass the form properties to as a parameter to you action execution method. See the Integration section for further details.

After performing all additional checks on the input values, the `commit()` method allows you to store them in the form data:

```
properties.commit();
```

Form Data

When valid form properties are commited, they are used to populate a form data instance.

The `de.odysseus.calyxo.forms.FormData` interface provides the methods `_getProperty(String name)` and `_setProperty(String name, Object value)` to access its properties.

The `FormsSupport` instance provides methods to access, remove or re-create saved form data for some action. This is useful if you want to initialize form data before (re)displaying the form:

```
FormData data = support.getFormData(request, "/checkin", true);
data._setProperty("arrival", new Date());
```

The way how form data is associated with your actions varies, depending on your environment. See the Integration section for further details.

2.9. Tag Library

The previous pages illustrated some important concepts of the *Calyxo Forms* component with a focus on configuration. We will now have a look at the way the *Calyxo Forms* component supports you, when writing your JSP pages. It offers an extensive set of JSP tags, designed to integrate the *Calyxo Forms* features with a minimal effort.

In the table below you find a list of commonly used tags from the *Calyxo Forms* tag library (a complete list is available in the reference section).

JSP Tags

Tag	Description
`form`	Defines an HTML form.
`text`	Renders a text input field.
`checkbox`	Renders a checkbox input field.
`radio`	Renders a radio button input field.
`group`	Container for checkbox or radio items.
`checkboxitem`	Renders a checkbox input field contained in a group.
`radioitem`	Renders a radio button input field contained in a group.
`select`	Renders a menu or selection list.
`option`	Renders a select option.

In the following paragraphs we will give an overview of the tags' attributes and look at a few simple examples illustrating the usage of these tags. The next page will then examine a more sophisticated approach for selections (incl. checkboxes and radio buttons) using list models to deliver dynamic option values instead of the static examples shown below.

Attributes

The exact list of attributes for a given tag together with a short description can be found in the reference section. As a rule of thumb we can say: the attribute list for one of the above tags contains the attributes of the corresponding HTML tag plus a core set of *Calyxo Forms* specific attributes.

All input tags (e.g. the `<text>`, `<checkbox>`, `<radio>`, or `<select>` tags) have to be embedded in a `<form>` tag and do require the `name` attribute, which defines the inputs's name (thus linking it to the corresponding input validations in the configuration file):

Attribute	Description
`name`	Required - Mapped to the HTML attribute of the same name.

Almost all of the HTML attributes are simply passed through (exceptions occur, when form data is available and used to override given default values, as is the case for the `checked` attribute of the `<checkbox>` or `<radio>` tags.)

There are four *Calyxo Forms* specific attributes, which are available for almost all *Calyxo Forms* tags. They are used to define the appearance of input fields, whose values did not pass the configured validation. One can separately define the `class` or `style` of input fields with data violating a single-field validation (validator) or a multiple-field validation (assertion). This is done by using one of the following four attributes.

Attribute	Description
`errorClass`	Appended to the HTML `class` attribute, if the validation of this input field failed.
`errorStyle`	Appended to the HTML `style` attribute, if the validation of this input field failed.
`assertClass`	Appended to the HTML `class` attribute, if an assertion concerning this input field failed.
`assertStyle`	Appended to the HTML `style` attribute, if an assertion concerning this input field failed.

If these attributes are not used for an input field, their values are derived from the embedding form (i.e., either a defined attribute value from the form is used or the form's default attribute value, which does result in a red background for failed single-field validations and in an orange background for failed multiple-field validations).

Forms

For the rest of this page we assume, that the *Calyxo Forms* tag library has been made available by declaring the namespace `http://calyxo.odysseus.de/jsp/forms`. In a JSP document, this is done by associating that namespace with a prefix as in:

```
<jsp:root xmlns:forms="http://calyxo.odysseus.de/jsp/forms" ... >
```

Alternatively, since the tags are only used inside an embedding `<form>` tag, the namespace declaration could be located inside the `<form>` tag as well:

```
<forms:form xmlns:forms="http://calyxo.odysseus.de/jsp/forms" ... >
```

Anyway, as a result all of the above tags will to be used with a `forms:` prefix.

In the following examples we assume, that the embedding form of the used input tag is linked to the embedding form of the field validation as described. This link is established as follows:

1. The form tag's `action` attribute connects the form to the action, that will be invoked when submitting the form:

   ```
   <forms:form action="/foo" ...> ... </forms:form>
   ```

2. The corresponding action configuation associates its inputs to a *Calyxo Forms* form definition named `bar` in a way depending on the controller implementation:
   - In a *Calyxo Control* action configuration, this is done in the `forms` filter's `form` parameter:

```
<action path="/foo" ...>
  <filter name="forms">
    <param name="form" value="bar">
    ...
  </filter>
  ...
</action>
```

- In a Struts action configuration, this is done by specifying a form bean whose name is equal to the name of the form definition to be used.

  ```
  <action path="/foo" input="bar" ...>
    ...
  </action>
  ```

  Please refer to the section on Integration for further details on *Calyxo Forms* related controller configuration.

We are now ready to focus on some of the usage scenarious for the various input tags provided by the *Calyxo Forms* tag library.

Text Fields

The most frequently used input fields are text fields, so this is a natural starting point for our tag examples. Here you see the `<text>` tag in action:

```
Please enter your departure date: <forms:text name="departure"/>
```

As mentioned, the above line has to be embedded in a `<form>` tag. The corresponding lines from your configuration file could look like this:

```
<field property="departure">
  <convert name="date">
    <message>
      <arg name="field" value="departure"/>
    </message>
  </convert>
</field>
```

Now your input field is linked to the corresponding field validation via its name (and the correspondence of the embedding forms). As a result, the entered string will be converted to a date and formatted in a standard way, if the input was valid. In case it was invalid the specified message is displayed and the input field is marked by a red background.

As an advanced feature you may map the result of a set of text input fields into an **array result**, instead of separate scalar values. This is especially helpful when handling tabular data. It is achieved by assigning the same name to all of the affected text input fields and adding an `index` attribute, starting with zero:

```
Please enter your children's names:
1. <forms:text name="children" index="0"/>
```

```
2. <forms:text name="children" index="1"/>
3. <forms:text name="children" index="2"/>
```

Our configuration file reflects the array result by embedding the `<field>` element into an `<input>` element, which allows us to set the `array` attribute for this input field to `true`:

```
<input name="children" array="true">
  <field property="children">
    <convert name="string"/>
  </field>
</input>
```

Checkboxes

We have three different approaches for checkboxes. The first one maps a **single checkbox** to a single boolean variable. It works as follows:

```
<forms:checkbox name="save"/>Please save my data for future visits.
```

The above tag from your JSP page is rendered into an HTML checkbox with attributes `name="save"` and `value="true"` (default). It is accompanied with a validation configuration like

```
<field property="save">
  <convert name="checkbox"/>
</field>
```

The `checkbox` converter converts the browser value (i.e., nothing or the string `"true"`) to a boolean value (`false` or `true`, respectively).

The second approach, using a so called **checkbox group**, is designed to implement a set of checkboxes (i.e., an input field where one can choose multiple answers). If we used the technique described above, we would end up with a set of boolean values (one with an own name for every checkbox). Alternatively, we may prefer an array result, containing a specific string for every checked checkbox. To achieve this, we use the `<group>` tag, which defines the name of the input variable and use nested `<checkboxitem>` tags to associate a specific string value with every checkbox:

```
Please choose your favorite color(s):
<forms:group name="colors">
  <forms:checkboxitem value="red"/>Red
  <forms:checkboxitem value="blue"/>Blue
  <forms:checkboxitem value="green"/>Green
  <forms:checkboxitem value="yellow"/>Yellow
</forms:group>
```

This group is rendered into four HTML checkboxes with attributes `name="colors"` and `value` as defined (the `value` attribute is required). The configuration file shows the conceptual change by setting the `array` attribute to `true` (which requires the use of the `input` element) and using the `string` converter for the array values.

```
<input name="colors" array="true">
  <field property="colors">
    <convert name="string"/>
  </field>
</input>
```

A third alternative is required, if you intend to implement a set of checkboxes, which shall map to an array result, but your JSP page structure doesn't allow the usage of the `<group>` tag, since it would overlap with another XML element (e.g. a table row or another checkbox group). In this case we use a more HTML-like approach, implementing every checkbox separately and linking them together via a common name attribute:

```
Please choose your favorite color(s):
<forms:checkbox name="colors" value="red"/>Red
<forms:checkbox name="colors" value="blue"/>Blue
<forms:checkbox name="colors" value="green"/>Green
<forms:checkbox name="colors" value="yellow"/>Yellow
```

Since this JSP fragment is rendered exactly as the checkbox group shown above, the configuration file for the string array result has not to be changed. Now one could ask why we suggest to use the `<group>` tag, if possible. Well, there are mainly two reasons:

- It extracts common attributes from the contained checkboxes. Not only the `name` attribute as seen above, but also the four attributes defining the validation error appearance.
- It is required to use list models, which deliver dynamic checkbox values, as explained on the selections page.

Radio Buttons

The usage of radio buttons shows some parallels to that of checkboxes, which we have seen in the previous section. There is no need for a single radio button, so we simply have to make our decision between two variants: with or without the `<group>` tag. First, let's look at typical **radio button group**, which is implemented using *Calyxo Forms*:

```
Please choose your favorite color:
<forms:group name="color">
  <forms:radioitem value="red" checked="checked"/>Red
  <forms:radioitem value="blue"/>Blue
  <forms:radioitem value="green"/>Green
  <forms:radioitem value="yellow"/>Yellow
</forms:group>
```

This group is rendered into four HTML radio buttons with attributes `name="color"` and `value` as defined (again, the `value` attribute is required). Since only a single selection is possible, there is no need to map the result into an array, so we end up with a single string containing the value of the chosen radio button. Thus, our configuration file looks like this:

```
<field property="color">
  <convert name="string"/>
```

```
</field>
```

As mentioned in the checkbox section, we might encounter situations where we would prefer an equivalent JSP page for the last example without grouping the radio buttons explicitly by using the `<group>` tag. Again, this can easily be done in a more HTML-like style:

```
Please choose your favorite color:
<forms:radio name="color" value="red" checked="checked"/>Red
<forms:radio name="color" value="blue"/>Blue
<forms:radio name="color" value="green"/>Green
<forms:radio name="color" value="yellow"/>Yellow
```

Since this JSP extract is rendered exactly like the grouped variant, the configuration file is not affected by the change.

Selections

Finally, we come to the selection lists. We illustrate the two main cases, single selection lists and multiple selection lists. Since the field's result type (scalar or array) is determined by the chosen list type, the difference between the two cases does not affect the JSP page, but only the configuration file. Let's look at an example:

```
Please choose a month:
<forms:select name="month">
  <forms:option value="1" selected="selected">January</forms:option>
  <forms:option value="2">February</forms:option>
  ...
  <forms:option value="12">December</forms:option>
</forms:select>
```

This JSP fragment could be used for a single selection list as well as a multiple selection list (the *Calyxo Forms* `<select>` tag has no `multiple` attribute, but generates the HTML attribute as required). We can turn the above example into a single selection list by defining a scalar result type with a configuration file entry like:

```
<field property="month">
  <convert name="integer"/>
</field>
```

This results in an HTML `<select>` tag without a `multiple` attribute, which is usually rendered as a drop-down list. If we prefer a real list instead of a drop-down list, we could have set the `size` attribute to a value of 5 for example, which would have been passed right through to the HTML `<select>` tag.

The next example illustrates a multiple selection list:

```
Please choose your favorite music styles:
<forms:select name="style" size="4">
  <forms:option value="1" selected="selected">Pop</forms:option>
```

```
  <forms:option value="2">Rock</forms:option>
  <forms:option value="3">Jazz</forms:option>
  <forms:option value="4">R&amp;B</forms:option>
  <forms:option value="5">Reggae</forms:option>
  <forms:option value="6">World</forms:option>
  <forms:option value="7">Classic</forms:option>
</forms:select>
```

Here we set the `size` attribute to a value of 4, but this is purely for presentational reasons, it's not the crucial difference, which makes the selection list multiple. This is found in the configuration file, where we declare an array result for the input variable called `style`:

```
<input name="style" array="true">
  <field property="style">
    <convert name="integer"/>
  </field>
</input>
```

There is one last advanced case left, which needs to be mentioned here: we can combine a set of single selection lists, so that their result values will end up in an array result instead of separate scalar values. What is this good for? It's the same reason, which already caused us to introduce arrays of text fields: the handling of tabular data. This leads to the same solution as for text fields: the single selection lists, which shall be grouped together, are assigned the same name and different index values:

```
Please enter your children's genders:
1. <forms:select name="gender" index="0">
     <forms:option value="F" selected="selected">Female</forms:option>
     <forms:option value="M">Male</forms:option>
   </forms:select>
2. <forms:select name="gender" index="1">
     <forms:option value="F" selected="selected">Female</forms:option>
     <forms:option value="M">Male</forms:option>
   </forms:select>
3. <forms:select name="gender" index="2">
     <forms:option value="F" selected="selected">Female</forms:option>
     <forms:option value="M">Male</forms:option>
   </forms:select>
```

The configuration file shows the corresponding result array:

```
<input name="gender" array="true">
  <field property="gender">
    <convert name="string"/>
  </field>
</input>
```

This results in three HTML `<select>` tags without a `multiple` attribute and with a `size` attribute of 1. (So the exact conditions for the generation of multiple selection lists are: the configuration file must define an array result **and** the JSP page's `<select>` tag must not have

an `index` attribute.)

2.10. Selections

Allowing users to select from a set of options is a common requirement in user interfaces.

- In a single selection, the selected subset is either empty or contains one element. A single selection may displayed as a group of radio buttons or a single selection list.
- In a multiple selection, any subset of valid options may occur. A multiple selection may be displayed as a group of check buttons or a multiple selection list.

To support selections, several aspects have to be taken into account:

- The set of valid options should be available through some kind of model.
- User input tags should be able to render themselves from a model.
- Validation and conversion of input values against a model should be supported.

List Models

Calyxo Forms introduces the concept of *list models*. A list model defines an ordered set of options, each option consisting of

- a *key* – a unique, non-empty string, which is taken as HTTP parameter value
- a *value* – a unique, non-null object, which is taken as form property value
- *labels* – localized strings to be displayed for this option

List models implement the `de.odysseus.calyxo.forms.view.ListModel` interface, which defines methods to map keys to values and vice versa, to iterate over option values in some particular order, to access labels, and so on.

Implementations

Though the `ListModel` interface may be implemented from scratch, the `de.odysseus.calyxo.forms.view.DefaultListModel` class is provided to serve as a base class for custom list model implementations. The default list model provides the following additional method to add an option for the specified key and value:

```
addOption(String key, Object value)
```

The `DefaultListModel` doesn't doesn't know how to lookup labels, so it simply uses list model keys as labels. Therefore, subclasses are probably left with the implemention of the following method:

```
String getLabel(Object value, Locale locale)
```

The `de.odysseus.calyxo.forms.view.I18nListModel` subclass implements that method by consulting an i18n support instance, taking its bundle property as a resource bundle name and using option keys as resource keys.

Configuration

A simple i18n-aware list model using the module's i18n support instance and bundle name `"choices"` could be created and saved to module scope like this:

```
ListModel model = new I18nListModel(moduleContext, "choices");
model.addOption("one", new Integer(1));
model.addOption("two", new Integer(2));
model.addOption("three", new Integer(3));
moduleContext.setAttribute("model123", model);
```

Alternatively, the same could be expressed in a configuration file:

```
<functions prefix="type" class="de.odysseus.calyxo.base.misc.TypeFunctions"/>

<set var="model123" scope="module">
  <object class="de.odysseus.calyxo.forms.view.I18nListModel">
    <base:constructor>
      <base:arg value="${moduleContext}"/>
      <base:arg value="choices"/>
    </base:constructor>
    <method name="addOption">
      <arg value="one"/>
      <arg value="${type:toInteger(1)}"/>
    </method>
    <method name="addOption">
      <arg value="two"/>
      <arg value="${type:toInteger(2)}"/>
    </method>
    <method name="addOption">
      <arg value="three"/>
      <arg value="${type:toInteger(3)}"/>
    </method>
  </object>
</set>
```

Note, that - if inside a *Calyxo Forms* configuration file - the above elements would have to be used with namespace prefix `base`.

Validation

Validating against a list model is done by mapping option keys to option values. The `de.odysseus.calyxo.forms.selection.ListModelConverter` is provided to do this job. The converter succeeds if the input string is a valid list model key. Additionally, a `null` or empty input is accepted and mapped to value `null`.

A single selection could be validated like this:

```
<input name="choice">
  <field property="choice">
    <convert name="list">
```

```
      <property name="attribute" value="model123"/>
      <property name="scope" value="module"/>
      <message>
        <arg name="field" value="choice"/>
      </message>
    </convert>
  </field>
</input>
```

A multiple selection has to be mapped to an array:

```
<input name="choices" array="true">
  <field property="choices">
    <convert name="list">
      <property name="attribute" value="model123"/>
      <property name="scope" value="module"/>
      <message>
        <arg name="field" value="choices"/>
      </message>
    </convert>
  </field>
</input>
```

JSP Tags

The *Calyxo Forms* tag library contains several tags, that support list models:

Radio/Checkbox Groups

The `<group>` tag may contain `<checkboxitem>` or `<radioitem>` tags to render checkbox- or radio groups. Nested group item elements inherit the `<group>`'s `name`, `class` and `style` attributes.

Consider a `<group name="foo">...</group>` tag. The `<group>` tag itself doesn't rendered to HTML, however

- nested `<checkboxitem value="...">` tags are rendered to `<input type="checkbox" name="foo" value="...">` HTML tags, resulting in a multiple (array) parameter `foo`.
- nested `<radioitem value="...">` tags are rendered to `<input type="radio" name="foo" value="...">` HTML tags, resulting in a single parameter `foo`.

The `forms.list` accessor can be used to iterate over list model values to render checkbox and radio groups:

```
<c:set var="model123" value="${calyxo.base.module.attribute['model123']}"/>
<c:set var="list" value="${calyxo.forms.list[model123]}"/>

<!-- render single selection as radio group -->
<forms:group name="choice" listModel="${model123}">
```

```
  <c:forEach items="${list.values['label']}" var="value">
    <forms:radioitem value="${list.key[value]}"/>
    <jsp:text> </jsp:text>${list.label[value]}<br/>
  </c:forEach>
</forms:group>

<!-- render multiple selection as checkbox group -->
<forms:group name="choices" listModel="${model123}">
  <c:forEach items="${list.values['label']}" var="value">
    <forms:checkboxitem value="${list.key[value]}"/>
    <jsp:text> </jsp:text>${list.label[value]}<br/>
  </c:forEach>
</forms:group>
```

When specifying a list model to the group using the `listModel` attribute (as above), then the `value` attributes of nested `<checkboxitem>` and `<radioitem>` tags are verified to be valid list model keys.

Selection Lists

The `<select>` tag may contain a `<listoptions>` tag to render all options from a list model at once. The list model has to be specified via the `listModel` attribute.

```
<c:set var="model123" value="${calyxo.base.module.attribute['model123']}"/>

<!-- render single selection -->
<forms:select name="choice" listModel="${model123}">
  <forms:listoptions sort="label"/>
</forms:select>

<!-- render multiple selection (use array input) -->
<forms:select name="choices" listModel="${model123}" size="3">
  <forms:listoptions sort="label"/>
</forms:select>
```

The `<listoption>` tag can be used to render a single list model option. The `value` attribute is used to specify a list option's key (not its value!). Thus, the previous example could be rewritten as

```
<c:set var="model123" value="${calyxo.base.module.attribute['model123']}"/>
<c:set var="list" value="${calyxo.forms.list[model123]}"/>

<!-- render single selection -->
<forms:select name="choice" listModel="${model123}">
  <c:forEach items="${list.values['label']}" var="value">
    <forms:listoption value="${list.key[value]}"/>
  </c:forEach>
</forms:select>
```

```
<!-- render multiple selection (use array input) -->
<forms:select name="choices" listModel="${model123}" size="3">
  <c:forEach items="${list.values['label']}" var="value">
    <forms:listoption value="${list.key[value]}"/>
  </c:forEach>
</forms:select>
```

However, to render all list model options, using the `<listoptions>` tag is recommended for simplicity and performance reasons.

By omitting the `value` attribute, the `<listoption>` tag can be used to render a default option:

```
<c:set var="model123" value="${calyxo.base.module.attribute['model123']}"/>

<forms:select name="choice" listModel="${model123}">
  <forms:listoption>Choose...</forms:listoption>
  <forms:listoptions sort="label"/>
</forms:select>
```

The default option will be selected in the list, if the group model selection is empty.

In the previous example, the *Choose...* option will be shown in the list, even if another option is selected. To avoid this, we need to check, if the group model has an empty selection:

```
<c:set var="model123" value="${calyxo.base.module.attribute['model123']}"/>

<forms:select name="choice" listModel="${model123}" varGroupModel="group">
  <c:if test="${group.selectedCount == 0}">
    <forms:listoption>Choose...</forms:listoption>
  </c:if>
  <forms:listoptions sort="label"/>
</forms:select>
```

2.11. I18n

The *Calyxo Forms* component supports locale-dependent variants of form definitions. This advanced feature may be used to define different validation variants, depending on the user's locale settings.

A locale specifies a *language* code, and may additionally specify a *country* code. If it does so, it may also specify a *variant* code (see class `java.util.Locale`). The *generalization* of a locale is defined by stripping off the last and least significant code. For example, generalizing `en_US` leads to `en`. Generalizing `en` leads to an unspecified locale.

Localized form definitions are collected into separate groups of `<forms>` elements. According to the locale properties, the `<forms>` element accepts attributes `language`, `country` and `variant` to specify a locale for the contained `<form>` elements.

...

3. Reference

3.1. Configuration

Throughout this reference, required attributes appear **strong**. Dynamic attributes (attributes, whose value may contains EL expressions) appear *emphasized*.

The elements described in the following sections are defined within namespace

```
http://calyxo.odysseus.de/xml/ns/forms
```

If the XML parser doesn't support XML Schema, DTD validation has to be used by declaring the *Calyxo Forms* document type as in:

```
<!DOCTYPE calyxo-forms-config
  PUBLIC "-//Odysseus Software GmbH//DTD Calyxo Forms 0.9//EN"
  "calyxo-forms-config.dtd">
```

Elements

Name	Description
`calyxo-forms-config`	Root element of a *Calyxo Forms* configuration file.
`validators`	Contains definitions of custom validators (i.e., matchers, checkers or converters) to be used in the <forms> section.
`matcher`	Defines a custom matcher to be used in the <forms> section.
`converter`	Defines a custom converter to be used in the <forms> section.
`checker`	Defines a custom checker to be used in the <forms> section.
`property`	Declares or sets a property value of a validator.
`message`	Specifies an error message.
`arg`	Sets an argument of an error message.
`forms`	Contains a set of form definitions (optionally for a specified locale).
`form`	Contains the validation rules concerning a single HTML input form.
`field`	Defines the sequence of validators (and an optional error message) applied to a single HTML input field.
`match`	Applies a matcher to an input field.
`convert`	Applies a converter to an input field.
`check`	Applies a checker to an input field.
`input`	...

assert	Defines an assertion rule concerning one or more input fields of a form.

3.1.1. The <calyxo-forms-config> Element

Purpose

The <calyxo-forms-config> element is the root element of a *Calyxo Forms* configuration file.

As common to all of *Calyxo*'s configuration files, the root element uses the xmlns attribute to specify the namespace and the version attribute to specify the XML schema/DTD version.

Attributes

Name	Type	Description
xmlns	CDATA	Required - XML namespace. Must be http://calyxo.odysseus.de/xml/ns/forms.
version	NMTOKEN	Required - DTD/Schema version number. Must be 0.9.

Body

The body of the <calyxo-forms-config> element is defined by the following sequence:

```
(base:import*, (base:functions | base:set | base:use)*, validators?, forms*)
```

The first four elements are common to all *Calyxo* components. They are described in the *Calyxo Base* configuration reference. The remaining two elements and their children are described in the following sections.

Related elements

<validators>, <forms>

3.1.2. The <validators> Element

Purpose

The <validators> element is used to define custom validators (i.e., matchers, checkers or converters) to be used in the <forms> section. In many cases the predefined *Calyxo Forms* validators will fulfill all your requirements and this element is not needed.

Attributes

The <validators> element has no attributes.

Body

The body of the <validators> element is defined by the following sequence:

```
(matcher | converter | checker)*
```

Related elements

<matcher>, <converter>, <checker>

3.1.3. The <matcher> Element

Purpose

The <matcher> element defines a custom matcher to be used in the <forms> section.

Attributes

Name	Type	Description
id	NMTOKEN	Required - Name of the matcher.
class	CDATA	Required & dynamic - Fully qualified Java class, which implements the de.odysseus.calyxo.forms.Matcher interface. (Examples can be found in the de.odysseus.calyxo.forms.match package.)

Body

The body of the <matcher> element is defined by the following sequence:

```
(property*, message?)
```

Related elements

<converter>, <checker>, <property>, <message>

3.1.4. The <converter> Element

Purpose

The <converter> element defines a custom converter to be used in the <forms> section.

Attributes

Name	Type	Description
id	NMTOKEN	Required - Name of the converter.
class	CDATA	Required & dynamic - Fully qualified Java class, which implements the de.odysseus.calyxo.forms.Converter interface. (Examples can be found in the de.odysseus.calyxo.forms.convert package.)

Body

The body of the <converter> element is defined by the following sequence:

```
(property*, message?)
```

Related elements

<matcher>, <checker>, <property>, <message>

3.1.5. The <checker> Element

Purpose

The <checker> element defines a custom checker to be used in the <forms> section.

Attributes

Name	Type	Description
id	NMTOKEN	Required - Name of the checker.
class	CDATA	Required & dynamic - Fully qualified Java class, which implements the de.odysseus.calyxo.forms.Checker interface. (Examples can be found in the de.odysseus.calyxo.forms.check package.)

Body

The body of the <checker> element is defined by the following sequence:

```
(property*, message?)
```

Related elements

<matcher>, <converter>, <property>, <message>

3.1.6. The <property> Element

Purpose

The <property> element declares or sets a property value of a validator. The usage differs depending on the context:

- In the <validators> section, where validators are declared, you have to declare all properties, which you intend to use in the <forms> section. This is done by adding a <property> element for every bean property of the underlying Java class, which shall be made available. If the value attribute is added, it acts as a default value, which may or may not be overridden when the validator is used. However, if the final attribute is set to true, the property value must not be overridden.
- In the <forms> section, where validators are used, you may override the value of one or more of the declared non-final properties of the validator by adding a <property> element. In this case both, the name and value, attributes are required.

A string property value will be automatically converted to the formal property type using de.odysseus.calyxo.base.util.ParseUtils. This will work for all primitive types, their wrapper classes and java.util.Date, provided the string value is given in standard notation. Finally, if the property type is javax.servlet.jsp.el.Expression, the string will be wrapped with "${" and "}" and parsed into an Expression (the expression may use functions declared in the validator's configuration file). For other property types, the property value must be an instance of that type.

Attributes

Name	Type	Description
name	NMTOKEN	Required - The name of the property.
value	CDATA	Dynamic - The value of the property. This attribute is required when overriding a validator property in a <match>, <convert> or <check> element.
final	true|false	States, if a validator property is final and must not be overridden. The default is false. Allowed only inside a <matcher>, <converter> or <checker> element.

Body

The <property> element has no body.

Related elements

<matcher>, <converter>, <checker>, <match>, <convert>, <check>

3.1.7. The <message> Element

Purpose

The <message> element specifies an error message. It is usually defined for a validator in the <validators> section, but may be overridden in the <forms> section, when a validator is applied to an input field.

Attributes

Name	Type	Description
key	NMTOKEN	The key under which the message is looked up in the specified bundle. (In most cases this attribute is a **required** one. It may only be omitted, if the <message> element's parent is a <match>, <convert> or <check> element.)
bundle	CDATA	Dynamic - The name of the bundle used to lookup the message.

Body

The body of the <message> element is defined by the following sequence:

```
(arg*)
```

Related elements

<arg>

3.1.8. The <arg> Element

Purpose

The <arg> element sets an argument of an error message.

A message argument may be specified by either of the following attribute combinations:

1. value to directly specify a value
2. property to take the value from a validator property
3. bundle and key to get the value from a resource

Attributes

Name	Type	Description
name	NMTOKEN	Name of the Argument. (This attribute may only be used in the <validators> section of the configuration file, not in the <forms> section.)
property	NMTOKEN	Takes the value of a property from the validator as argument

		value.
value	CDATA	Dynamic - Directly specifies the argument value.
key	NMTOKEN	Used in conjunction with bundle - Specifies the resource key containing the argument value.
bundle	CDATA	Dynamic - Used in conjunction with key - Specifies the resource bundle name to be used to lookup the argument value.

Body

The <arg> element has no body.

Related elements

<message>

3.1.9. The <forms> Element

Purpose

The <forms> element contains a set of form definitions (optionally for a specified locale).

Attributes

Name	Type	Description
language	NMTOKEN	The language code of the locale, which applies to all forms in this <forms> section.
country	NMTOKEN	The country code of the locale, which applies to all forms in this <forms> section. (Setting the country attribute does only make sense, if the language is also specified.)
variant	NMTOKEN	The country code of the locale, which applies to all forms in this <forms> section. (Setting the variant attribute does only make sense, if language and country are also specified.)

Body

The body of the <forms> element is defined by the following sequence:

(form*)

Related elements

<form>

3.1.10. The <form> Element

Purpose

The <form> element contains the validation rules concerning a single HTML input form.

Attributes

Name	Type	Description
name	NMTOKEN	The name of the form.

Body

The body of the <form> element is defined by the following sequence:

```
((field | input)*, assert*)
```

Related elements

<forms>, <field>, <input>, <assert>

3.1.11. The <field> Element

Purpose

The <field> element ...

Attributes

Name	Type	Description
property	NMTOKEN	Required - Specifies the name of a form data property, this field maps to. If the field is a direct child of the <form> tag, this attribute also specifies the name of the HTTP input parameter.
null	CDATA	Dynamic - Specifies an object, that should be taken as form data value if an input value is converted to null.

Body

The body of the <field> element is defined by the following sequence:

```
(match*, convert?, check*, message?)
```

Related elements

<match>, <convert>, <check>, <message>

3.1.12. The <match> Element

Purpose

The <match> element applies a matcher to an input field.

Attributes

Name	Type	Description
name	NMTOKEN	Required - Name of the matcher.

Body

The body of the <match> element is defined by the following sequence:

```
(property*, message?)
```

Related elements

<convert>, <check>, <property>, <message>

3.1.13. The <convert> Element

Purpose

The <convert> element applies a converter to an input field.

Attributes

Name	Type	Description
name	NMTOKEN	Required - Name of the converter.

Body

The body of the <convert> element is defined by the following sequence:

```
(property*, message?)
```

Related elements

<match>, <check>, <property>, <message>

3.1.14. The <check> Element

Purpose

The <check> element applies a checker to an input field.

Attributes

Name	Type	Description
name	NMTOKEN	Required - Name of the checker.

Body

The body of the <check> element is defined by the following sequence:

```
(property*, message?)
```

Related elements

<match>, <convert>, <property>, <message>

3.1.15. The <input> Element

Purpose

The <input> element ...

Attributes

Name	Type	Description
name	NMTOKEN	Required - Specifies the HTTP parameter name.
array	true|false	Set to true for an array input.
relax	CDATA	Dynamic - A boolean expression (without leading '${' and trailing '}'); if specified, the expression will be evaluated when validating the input has failed. If it evaluates to true, no error message will be generated. However, the form is still invalid.
ignore	CDATA	Dynamic - A boolean expression (without leading '${' and trailing '}'); if specified, the expression will be evaluated before validating the input. If it evaluates to true, validation will be skipped for that input. If a form is valid or not does not depend on ignored inputs.

Body

The body of the <input> element is defined by the following sequence:

```
(field+, message?)
```

Related elements

<field>, <message>

3.1.16. The <assert> Element

Purpose

The <assert> element defines an assertion rule concerning one or more input fields of a form.

Attributes

Name	Type	Description
test	CDATA	Required, Dynamic - A boolean expression that evaluates to true, if the inputs pass the test successfully.

Body

The body of the <assert> element is defined by the following sequence:

```
(message)
```

Related elements

<message>

3.2. Accessors

The ${calyxo.forms} accessors provides access to data related to the *Calyxo Forms* component. They are automatically registered to the *Calyxo Base*' access support when the module is loaded.

To make the accessors available in JSP pages, the <base:access> tag is used to install the access tree into request scope:

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0"
  xmlns:base="http://calyxo.odysseus.de/jsp/base">
    ...
```

```
    <base:access var="calyxo"/>
    ...
</jsp:root>
```

In our examples, we assume, that the accessors have already been installed at request attribute `calyxo`.

3.2.1. The forms.data accessor

data[action]

Answers the form data for the specified action path by delegating to `FormsSupport.getFormData()`. The result is `null` if no form data was found. Otherwise, the object wrapped by the form data instance - usually a map or bean - is returned.

Example

The expression `${calyxo.forms.data['/foo']}` searches the form data instance stored for action `'/foo'`.

3.2.2. The forms.list accessor

list[model]

Answers an object that accesses the specified list model.

- `label[`*value*`]` – get the label for the specified list model value; delegates to `ListModel.getLabel(...)`
- `value[`*key*`]` – get the list model value for the specified key; delegates to `ListModel.getValue(...)`
- `key[`*value*`]` – get the key for the specified list model value; delegates to `ListModel.getKey(...)`
- `values[`*criteria*`]` – get an iterator over the sorted values of the list model; *criteria* must match the regular expression `(+|-)?(index|value|key|label)`; delegates to `ListModel.getValues(...)`

Examples

The expression `${calyxo.forms.list[foo].label[bar]}` finds the list model at attribute `foo` and answers the label for the list model value found at attribute `bar`.

3.3. Tag Library

The *Calyxo Forms* custom tag library contains tags to render input fields. In a JSP file, just associate the prefix you want to use for the tags with URI

`http://calyxo.odysseus.de/jsp/forms`:

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0"
  xmlns:forms="http://calyxo.odysseus.de/jsp/forms">
  ...
</jsp:root>
```

Since the tag library descriptor is contained in the *Calyxo Forms* jar file, it is already available to applications. The container will automatically find it. For documentation purposes, a copy is located in `CALYXO_HOME/calyxo-forms/conf/share/calyxo-forms.tld`.

All attributes may be expressed using **runtime expressions**. Most of the attributes are **optional**. If an attribute is **required** for a specific tag, this is mentioned in the corresponding attribute description (and indicated by an attribute name printed in bold).

General Tags

Name	Description
`debug`	Renders an HTML table showing the current form validation result.

HTML Tags

Name	Description
`form`	Defines an HTML form.
`text`	Renders a text input field.
`password`	Renders a password input field.
`hidden`	Renders a hidden input field.
`textarea`	Renders a textarea input field.
`checkbox`	Renders a checkbox input field.
`radio`	Renders a radio button input field.
`group`	Container for checkbox or radio items.
`checkboxitem`	Renders a checkbox input field contained in a group.
`radioitem`	Renders a radio button input field contained in a group.
`select`	Renders a menu or selection list.
`option`	Renders a select option.
`listoption`	Renders a select option obtained from a list model.
`listoptions`	Renders a set of select options obtained from a list model.

Attribute groups

We use the following **abbreviations** to refer to groups of HTML attributes. The first three (`%coreattrs`, `%i18n`, `%events`) have the same meaning as in the HTML 4.01 specification, whereas the last one (`%input`) is only used within this documentation.

Abbreviation	HTML Attributes
`%coreattrs`	`id, class, style, title`
`%i18n`	`lang, dir`

`%events`	`onclick, ondblclick`
	`onmousedown, onmouseup, onmouseover, onmousemove, onmouseout`
	`onkeydown, onkeyup, onkeypress`

`%input`	`accesskey, alt, disabled, readonly, tabindex` `onfocus, onblur, onchange`

All of them are mapped to the HTML attribute of the same name. Usually their **values are not modified**, but there are two minor exceptions from this rule: the attributes `disabled` and `readonly`. If their value is set to an arbitrary string, this is mapped to an HTML attribute containing the attribute name (i.e., `disabled="disabled"` or `readonly="readonly"`).

3.3.1. The <debug> Tag

Purpose

The `<debug>` tag renders an HTML table showing the current form validation result.

Attributes

The `<debug>` tag has no attributes.

Body

The `<debug>` tag has no body.

3.3.2. The <form> Tag

Purpose

The `<form>` tag defines an HTML form (i.e., it is mapped to an HTML `<form>` element). It is an enhanced version of the `<base:form>` tag from the *Calyxo Base* component (with six additional attributes related to validation issues: `assertClass`, `assertStyle`, `errorClass`, `errorStyle` and `create`).

Attributes

Attribute groups: %coreattrs, %i18n and %events.

Name	Description
accept	Mapped to the HTML attribute of the same name.
action	Required - Module-relative path to the action in the current module, that should be invoked when submitting the form. The tag will transform this into a context-relative path, when it renders the HTML form tag's action attribute.
assertClass	Appended to the HTML class attribute of a contained input field, if an assertion concerning that input field failed and if the equally named attribute of the input field is not set.
assertStyle	Appended to the HTML style attribute of a contained input field, if an assertion concerning that input field failed and if the equally named attribute of the input field is not set. (Default: "background-color: orange;", but this value is only used if neither the contained input field nor this form have an assertClass attribute value set.)
create	If given, must be true or false. The form tag uses the FormsSupport.getFormData(...) method to retrieve the form data instance for its action path. The boolean create parameter instructs the forms support to lazily create a new form data instance, if none is present for the specified path. (Default: false.)
enctype	Mapped to the HTML attribute of the same name.
errorClass	Appended to the HTML class attribute of a contained input field, if the validation of that input field failed and if the equally named attribute of the input field is not set.
errorStyle	Appended to the HTML style attribute of a contained input field, if the validation of that input field failed and if the equally named attribute of the input field is not set. (Default: "background-color: red;", but this value is only used if neither the contained input field nor this form have an errorClass attribute value set.)
method	Mapped to the HTML attribute of the same name.
name	Mapped to the HTML attribute of the same name.
onreset	Mapped to the HTML attribute of the same name.
onsubmit	Mapped to the HTML attribute of the same name.
target	Mapped to the HTML attribute of the same name.

Body

The <form> tag requires a body containing the contents of the form (i.e., its input fields, buttons, etc.).

3.3.3. The <text> Tag

Purpose

The <text> tag renders a text input field (i.e., it is mapped to an HTML <input> element with type="text").

Requirements

A <text> tag has to be embedded in a <form> tag.

Attributes

Attribute groups: %coreattrs, %i18n, %events and %input.

Name	Description
assertClass	Appended to the HTML class attribute, if an assertion concerning this input field failed. (Default: equally named attribute value of the embedding form.)
assertStyle	Appended to the HTML style attribute, if an assertion concerning this input field failed. (Default: equally named attribute value of the embedding form.)
errorClass	Appended to the HTML class attribute, if the validation of this input field failed. (Default: equally named attribute value of the embedding form.)
errorStyle	Appended to the HTML style attribute, if the validation of this input field failed. (Default: equally named attribute value of the embedding form.)
index	Required for array inputs - The array index for this input field.
maxlength	Mapped to the HTML attribute of the same name.
name	Required - Mapped to the HTML attribute of the same name.
onselect	Mapped to the HTML attribute of the same name.
size	Mapped to the HTML attribute of the same name.
value	Mapped to the HTML attribute of the same name, if no current value is available.

Body

The <text> tag has no body.

Related tags

<password>, <hidden>, <textarea>

3.3.4. The <password> Tag

Purpose

The `<password>` tag renders a password input field (i.e., it is mapped to an HTML `<input>` element with `type="password"`).

Requirements

A `<password>` tag has to be embedded in a `<form>` tag.

Attributes

Attribute groups: `%coreattrs, %i18n, %events` and `%input`.

Name	Description
`assertClass`	Appended to the HTML `class` attribute, if an assertion concerning this input field failed. (Default: equally named attribute value of the embedding form.)
`assertStyle`	Appended to the HTML `style` attribute, if an assertion concerning this input field failed. (Default: equally named attribute value of the embedding form.)
`errorClass`	Appended to the HTML `class` attribute, if the validation of this input field failed. (Default: equally named attribute value of the embedding form.)
`errorStyle`	Appended to the HTML `style` attribute, if the validation of this input field failed. (Default: equally named attribute value of the embedding form.)
`index`	Required for array inputs - The array index for this input field.
`maxlength`	Mapped to the HTML attribute of the same name.
`name`	Required - Mapped to the HTML attribute of the same name.
`onselect`	Mapped to the HTML attribute of the same name.
`redisplay`	If set to `true`, the tag displays its value (using stars). Otherwise the input field appears empty. (Default: `false`.)
`size`	Mapped to the HTML attribute of the same name.
`value`	Mapped to the HTML attribute of the same name, if no current value is available.

Body

The `<password>` tag has no body.

Related tags

`<text>`, `<hidden>`, `<textarea>`

3.3.5. The <hidden> Tag

Purpose

The <hidden> tag renders a hidden input field (i.e., it is mapped to an HTML <input> element with type="hidden").

Requirements

A <hidden> tag has to be embedded in a <form> tag.

Attributes

Name	Description
index	Required for array inputs - The array index for this input field.
name	Required - Mapped to the HTML attribute of the same name.
value	Mapped to the HTML attribute of the same name, if no current value is available.

Body

The <hidden> tag has no body.

Related tags

<text>, <password>, <textarea>

3.3.6. The <textarea> Tag

Purpose

The <textarea> tag renders a textarea input field (i.e., it is mapped to an HTML <textarea> element).

Requirements

A <textarea> tag has to be embedded in a <form> tag.

Attributes

Attribute groups: %coreattrs, %i18n, %events and %input.

Name	Description
assertClass	Appended to the HTML class attribute, if an assertion concerning this input field failed. (Default: equally named attribute value of the embedding form.)
assertStyle	Appended to the HTML style attribute, if an assertion concerning this input field failed. (Default: equally named attribute value of the embedding form.)
cols	Required - Mapped to the HTML attribute of the same name.
errorClass	Appended to the HTML class attribute, if the validation of this input field failed. (Default: equally named attribute value of the embedding form.)
errorStyle	Appended to the HTML style attribute, if the validation of this input field failed. (Default: equally named attribute value of the embedding form.)
index	Required for array inputs - The array index for this input field.
name	Required - Mapped to the HTML attribute of the same name.
onselect	Mapped to the HTML attribute of the same name.
rows	Required - Mapped to the HTML attribute of the same name.

Body

The <textarea> tag has an optional body, which is used as an **initial contents** when the textarea input field is displayed for the first time and no form data is available to supply the initial data.

Related tags

<text>, <hidden>, <password>

3.3.7. The <checkbox> Tag

Purpose

The <checkbox> tag renders a checkbox input field (i.e., it is mapped to an HTML <input> element with type="checkbox").

Requirements

A <checkbox> tag has to be embedded in a <form> tag.

Attributes

Attribute groups: %coreattrs, %i18n, %events and %input.

Name	Description
assertClass	Appended to the HTML class attribute, if an assertion concerning this input field failed. (Default: equally named attribute value of the embedding form.)
assertStyle	Appended to the HTML style attribute, if an assertion concerning this input field failed. (Default: equally named attribute value of the embedding form.)
checked	Mapped to the HTML attribute of the same name, if no current value is available.
errorClass	Appended to the HTML class attribute, if the validation of this input field failed. (Default: equally named attribute value of the embedding form.)
errorStyle	Appended to the HTML style attribute, if the validation of this input field failed. (Default: equally named attribute value of the embedding form.)
name	Required - Mapped to the HTML attribute of the same name.
value	Mapped to the HTML attribute of the same name (default value is "true").

Body

The <checkbox> tag has no body.

Related tags

<checkboxitem>, <radio>, <radioitem>

3.3.8. The <radio> Tag

Purpose

The <radio> tag renders a radio button input field (i.e., it is mapped to an HTML <input> element with type="radio").

Requirements

A <radio> tag has to be embedded in a <form> tag.

Attributes

Attribute groups: %coreattrs, %i18n, %events and %input.

Name	Description
assertClass	Appended to the HTML class attribute, if an assertion concerning this input field failed. (Default: equally named attribute value of the embedding form.)
assertStyle	Appended to the HTML style attribute, if an assertion concerning this input field failed. (Default: equally named attribute value of the embedding form.)
checked	Mapped to the HTML attribute of the same name, if no current value is available.
errorClass	Appended to the HTML class attribute, if the validation of this input field failed. (Default: equally named attribute value of the embedding form.)
errorStyle	Appended to the HTML style attribute, if the validation of this input field failed. (Default: equally named attribute value of the embedding form.)
name	Required - Mapped to the HTML attribute of the same name.
value	Required - Mapped to the HTML attribute of the same name.

Body

The <radio> tag has no body.

Related tags

<radioitem>, <checkbox>, <checkboxitem>

3.3.9. The <group> Tag

Purpose

The <group> tag does not render to a specific element. It is a container for two or more checkbox or radio items, which appear in its body. We use a group to extract the redundant attributes (especially the name attribute) from a set of checkbox/radio items corresponding to the same input field.

A <group> tag may expose its group model to page scope. The group model contains information about the selection state. If form data is available, it "knows" how many and which items are to be checked.

The <group> tag may take a list model in its listModel attribute. In this case, the values of contained checkbox and radio items are verified to be valid keys in that list model.

Requirements

A <group> tag has to be embedded in a <form> tag.

Attributes

Attribute groups: none.

Name	Description
`assertClass`	Appended to the HTML `class` attribute of the contained items, if an assertion concerning this input group failed. (Default: equally named attribute value of the embedding form.)
`assertStyle`	Appended to the HTML `style` attribute of the contained items, if an assertion concerning this input group failed. (Default: equally named attribute value of the embedding form.)
`errorClass`	Appended to the HTML `class` attribute of the contained items, if the validation of this input group failed. (Default: equally named attribute value of the embedding form.)
`errorStyle`	Appended to the HTML `style` attribute of the contained items, if the validation of this input group failed. (Default: equally named attribute value of the embedding form.)
`listModel`	The group's list model. If given, the `value` attributes of contained checkbox and radio items are verified to to be valid keys of that model.
`name`	Required - Used as the `name` attribute for the contained items.
`varGroupModel`	If set, the tag exposes its group model to a page scope attribute of that name. The group model contains information about the selection state.

Body

The `<group>` tag requires a body containing the checkbox or radio items, which are to be grouped together.

Related tags

`<checkboxitem>`, `<radioitem>`

3.3.10. The <checkboxitem> Tag

Purpose

The `<checkboxitem>` tag renders a checkbox input field (i.e., it is mapped to an HTML `<input>` element with `type="checkbox"`).

Requirements

A `<checkboxitem>` tag has to be embedded in a `<group>` tag.

Attributes

Attribute groups: %coreattrs, %i18n, %events and %input.

Name	Description
checked	Mapped to the HTML attribute of the same name, if no current value is available.
value	Required - Mapped to the HTML attribute of the same name. The value has to be a valid key in the group model of the enclosing <group> tag.

Body

The <checkboxitem> tag has no body.

Related tags

<checkbox>, <radioitem>, <radio>

3.3.11. The <radioitem> Tag

Purpose

The <radioitem> tag renders a radio button input field (i.e., it is mapped to an HTML <input> element with type="radio").

Requirements

A <radioitem> tag has to be embedded in a <group> tag.

Attributes

Attribute groups: %coreattrs, %i18n, %events and %input.

Name	Description
checked	Mapped to the HTML attribute of the same name, if no current value is available.
value	Required - Mapped to the HTML attribute of the same name. The value has to be a valid key in the group model of the enclosing <group> tag.

Body

The <radioitem> tag has no body.

Related tags

<radio>, <checkboxitem>, <checkbox>

3.3.12. The <select> Tag

Purpose

The <select> tag renders a menu or selection list (i.e., it is mapped to an HTML <select> element).

A <select> tag may expose its group model to page scope. The group model contains information about the selection state. If form data is available, it "knows" how many and which options are to be selected.

The <select> tag may take a list model in its listModel attribute. In this case, the values of contained options are verified to be valid keys in that model.

A <select> tag is used

1. as a *single selection*, if the tag correponds to a non-array input or if it corresponds to an array input but specifies an index via its index attribute.
2. as a *multiple selection*, if the tag corresponds to an array input and doesn't specify an index.

Requirements

A <select> tag has to be embedded in a <form> tag.

Attributes

Attribute groups: %coreattrs, %i18n and %events.

Name	Description
assertClass	Appended to the HTML class attribute, if an assertion concerning this input field failed. (Default: equally named attribute value of the embedding form.)
assertStyle	Appended to the HTML style attribute, if an assertion concerning this input field failed. (Default: equally named attribute value of the embedding form.)
disabled	Mapped to the HTML attribute of the same name.
errorClass	Appended to the HTML class attribute, if the validation of this input field failed. (Default: equally named attribute value of the embedding form.)
errorStyle	Appended to the HTML style attribute, if the validation of this input field failed. (Default: equally named attribute value of the embedding form.)
index	For array inputs - The array index for this input field.
listModel	The select's list model. If given, the value attributes of contained options

	are verified to to be valid keys in that model.
name	Required - Mapped to the HTML attribute of the same name.
onblur	Mapped to the HTML attribute of the same name.
onchange	Mapped to the HTML attribute of the same name.
onfocus	Mapped to the HTML attribute of the same name.
size	Mapped to the HTML attribute of the same name.
tabindex	Mapped to the HTML attribute of the same name.
varGroupModel	If set, the tag exposes its group model to a page scope attribute of that name. The group model contains information about the selection state.

Body

The <select> tag requires a body containing the <option>, <listoption> or <listoptions> tags, which define the entries of the rendered select element.

Related tags

<option>, <listoption>, <listoptions>

3.3.13. The <option> Tag

Purpose

The <option> tag renders a select option (i.e., it is mapped to an HTML <option> element).

Requirements

An <option> tag has to be embedded in a <select> tag.

Attributes

Attribute groups: %coreattrs, %i18n and %events.

Name	Description
disabled	Mapped to the HTML attribute of the same name.
selected	Mapped to the HTML attribute of the same name, if no current value is available.
value	Required - Mapped to the HTML attribute of the same name. The value has to be a valid key in the group model of the enclosing <select> tag.

Body

The <option> tag requires a body containing the label for the entry in the selection list.

Related tags

<select>, <listoption>, <listoptions>

3.3.14. The <listoption> Tag

Purpose

The <listoption> tag renders a select option (i.e., it is mapped to an HTML <option> element). The option's label is obtained from the list model, which has been attached to the enclosing <select> tag.

Requirements

A <listoption> tag has to be embedded in a <select> tag with an attached list model.

Attributes

Attribute groups: %coreattrs, %i18n and %events.

Name	Description
disabled	Mapped to the HTML attribute of the same name.
value	Mapped to the HTML attribute of the same name. If set, its value has to be a valid key in the list model attached to the enclosing <select> tag. If no value attribute is specified, the option is rendered as a "default option": the value attribute of the rendered HTML <option> element will be set to the empty string and the label will be taken from the tag body. The HTML option will be selected if the selection of the <select> tag's group model is empty.

Body

If the value attribute is set, the option's label is taken from the group list model and the body is ignored. Otherwise, if the tag is to be rendered as a default option, the body is taken as the label for the entry in the selection list.

Related tags

<select>, <listoptions>, <option>

3.3.15. The <listoptions> Tag

Purpose

The <listoptions> tag renders a set of select options obtained from a list model, which has been specified in the enclosing <select> tag.

Requirements

A <listoptions> tag has to be embedded in a <select> tag with an attached list model.

Attributes

Attribute groups: %coreattrs, %i18n and %events.

Name	Description
disabled	Mapped to the HTML attribute of the same name.
selection	May be used to specify one or more list model keys to be used as a default selection when no current selection is available. The specified value must of type String or String[].
sort	Determines the sorting parameter used to bring the rendered option entries into an appropriate order. Possible values are • "index" - Default. The natural order defined by the list model. • "key" - The entries are ordered by their key strings (lexicographically). • "value" - The entries are ordered by their value objects (as comparables). • "label" - The entries are ordered by their label strings (lexicographically). Each of the above choices may be preceded by a minus sign (for example "-label") to receive a descending order instead of an ascending one.

Body

The <listoptions> tag has no body.

Related tags

<select>, <listoption>, <option>

3.4. Predefined Validators

The *Calyxo Forms* component contains predefined validators (i.e., matchers, converters and checkers), which are suitable for various standard tasks arising in the context of input validation. Among others, there are matchers, which trim the input or match it against a regular expression, converters for all of the standard data types, and checkers for string length validation or numerical ranges.

Since the configuration file containing the declarations of the predefined validators is contained in the *Calyxo Forms* jar file, all of them are already available to applications. For documentation purposes, a copy is located in `CALYXO_HOME/calyxo-forms/conf/share/calyxo-validators.xml`.

In the same directory you can find a copy of the default messages contained in the file `calyxo-validators.properties`. All validators, which can fail, have a predefined error message to be found in this file. Every predefined error message takes an argument used to identify the input field which caused the error. It is named `field` in all validator message declarations. A message belonging to a validator with one or more properties may have additional unnamed arguments, which are set to current property values of the validator, to make the message more precise. However, these message arguments cannot be overridden.

Therefore, using a predefined validator typically means to supply the message argument named `field` as in

```
<check name="less">
  <message>
    <arg name="field" .../>
  </message>
</check>
```

See the section on Messages on how argument values are specified.

If you intend to use a different message bundle, e.g. `myBundle`, you have two options:

1. To use your own bundle for all predefined validator messages, invoke `setBundleAlias("calyxo-forms-validators", "myBundle")` on the module's `I18nSupport` instance. There's a convenient way to do this in your configuration file:

   ```
   <calyxo-forms-config version="0.9"
     xmlns:base="http://calyxo.odysseus.de/xml/ns/base"
     xmlns="http://calyxo.odysseus.de/xml/ns/forms">
     ...
     <base:use>
       <base:member class="de.odysseus.calyxo.base.I18nSupport">
         <base:method name="getInstance">
           <base:arg value="${moduleContext}"/>
         </base:method>
       </base:member>
       <base:method name="setBundleAlias">
         <base:arg value="calyxo-forms-validators"/>
         <base:arg value="myBundle"/>
       </base:method>
     </base:use>
     ...
   </calyxo-forms-config>
   ```

2. To use a different bundle in a single validation, override the `bundle` attribute in the `message` element.

```
<convert name="date">
  <message bundle="myBundle">
    <arg name="field" .../>
  </message>
</convert>
```

To replace the whole message, specify both the `bundle` and `key` attribute in the `message` element. For example, this may be desired when using the `regexp` matcher, which produces a rather generic message including the the regular expression pattern:

```
<match name="regexp">
  <property name="pattern" value="..."/>
  <message bundle="myBundle" key="myKey">
    ...
  </message>
</check>
```

On the following pages you can find an overview describing all the predefined validators from the *Calyxo Forms* component.

3.4.1. Predefined Matchers

Below you find an overview describing the predefined matchers from the *Calyxo Forms* component. All of them implement the `de.odysseus.calyxo.forms.Matcher` interface and can be found in the `de.odysseus.calyxo.forms.match` package.

Name	Class	Description
`notEmpty`	`NotEmptyMatcher`	Matches the complete input string if its length is larger than zero. Otherwise the matcher fails. It has no properties.
`regexp`	`RegExpMatcher`	Tries to match the input with a regular expression. Returns the matched portion of the input, if successful. Property: • `pattern` - The regular expression.
`trim`	`TrimMatcher`	Strips off leading and trailing white space. It can not fail and has no properties.

3.4.2. Predefined Converters

Below you find an overview describing the predefined converters from the *Calyxo Forms* component. All of them implement the `de.odysseus.calyxo.forms.Converter` interface and can be found in the `de.odysseus.calyxo.forms.convert` package.

Name	Class	Description
`bigDecimal`	`BigDecimalConverter`	Parses and formats objects of type `BigDecimal`. It has no properties.
`bigInteger`	`BigIntegerConverter`	Parses and formats objects of type `BigInteger`.

		Property: • `radix` - The radix of the representation (from `2` to `36`). (Default value is `10`.)
`boolean`	`BooleanConverter`	Parses and formats objects of type `Boolean`. Properties: • `trueString` - The string which represents a boolean value of `true`. (Default value is `"true"`.) • `falseString` - The string which represents a boolean value of `false`. (Default value is `"false"`.) • `default` - Default value (in case of empty input, defaults to `null`.)
`byte`	`ByteConverter`	Parses and formats objects of type `Byte`. Properties: • `default` - Default value (in case of empty input, defaults to `null`.) • `groupingUsed` - Boolean value determining whether grouping is turned on when formatting the value. (Default value is `true`.)
`calendar`	`CalendarConverter`	Parses and formats objects of type `Calendar`. It has no properties.
`checkbox`	`BooleanConverter`	Convenience converter to be used with checkbox inputs. Parses and formats objects of type `Boolean`. Empty inputs are parsed to `false`. Properties: • `trueString` - The string which represents a boolean value of `true`. (The checkbox value, default is `"true"`.)
`date`	`DateConverter`	Parses and formats objects of type `Date` using a 'short date format'. It has no properties.
`double`	`DoubleConverter`	Parses and formats objects of type `Double`. Properties: • `default` - Default value (in case of empty input, defaults to `null`.) • `groupingUsed` - Boolean value determining whether grouping is turned on when formatting the value. (Default value is `true`.) • `minimumFractionDigits` - Passed through to `java.text.NumberFormat`. • `maximumFractionDigits` - Passed through to `java.text.NumberFormat`.
`float`	`FloatConverter`	Parses and formats objects of type `Float`. Properties: • `default` - Default value (in case of empty input, defaults to `null`.)

		• groupingUsed - Boolean value determining whether grouping is turned on when formatting the value. (Default value is true.) • minimumFractionDigits - Passed through to java.text.NumberFormat. • maximumFractionDigits - Passed through to java.text.NumberFormat.
integer	IntegerConverter	Parses and formats objects of type Integer. Properties: • default - Default value (in case of empty input, defaults to null.) • groupingUsed - Boolean value determining whether grouping is turned on when formatting the value. (Default value is true.)
list	ListModelConverter	Parses and formats objects contained in a ListModel. The list model is located via the scope, attribute and property properties. Properties: • scope - Attribute scope (Default value is module.) • attribute - Attribute name • property - Property of the object located by scope and attribute containing the list model (optional).
long	LongConverter	Parses and formats objects of type Long. Properties: • default - Default value (in case of empty input, defaults to null.) • groupingUsed - Boolean value determining whether grouping is turned on when formatting the value. (Default value is true.)
short	ShortConverter	Parses and formats objects of type Short. Properties: • default - Default value (in case of empty input, defaults to null.) • groupingUsed - Boolean value determining whether grouping is turned on when formatting the value. (Default value is true.)
string	StringConverter	Parses and formats objects of type String. It can not fail and has one property: • default - Default value (in case of empty input, defaults to null.)
time	TimeConverter	Parses and formats objects of type Date using a 'short time format'. It has no properties.

3.4.3. Predefined Checkers

Below you find an overview describing all of the predefined checkers from the *Calyxo Forms* component. All of them implement the `de.odysseus.calyxo.forms.Checker` interface and can be found in the `de.odysseus.calyxo.forms.check` package.

Name	**Class**	**Description**
`notNull`	`NotNullChecker`	Accepts any input different from `null`. It has no properties.
`length`	`LengthChecker`	Checks the length of a string to be in the range specified by its properties, which are: • `min` • `max`
`el`	`ELChecker`	Checks that a given EL expression evaluates to `true`. The expression refers to the value via the variable `property` (other variables, which are resolved as usual are `requestScope`, `sessionScope`, `moduleScope`, `applicationScope`, `param` and `moduleContext`). Property: • `expression` - EL expression (without surrounding `"${"` and `"}"`).
`interval`	`RangeChecker`	Checks a numerical value to be in the range specified by its properties, which are: • `min` • `allowMin` - Boolean value determining, whether the lower border is contained in the interval or not. (Default value is `true`.) • `max` • `allowMax` - Boolean value determining, whether the upper border is contained in the interval or not. (Default value is `true`.)
`less`	`RangeChecker`	Checks a numerical value to be less than the specified property value: • `max`
`most`	`RangeChecker`	Checks a numerical value to be less than or equal to the specified property value: • `max`
`greater`	`RangeChecker`	Checks a numerical value to be greater than the specified property value: • `min`
`least`	`RangeChecker`	Checks a numerical value to be greater than or equal to the specified property value: • `min`

4. Extension Points

4.1. Adding your own validators

Note

The following guide for building custom validators assumes that you are familiar with the notion of *Calyxo Forms* validators (i.e., matchers, converters and checkers) as it is illustrated in the concepts section.

Before writing your own validator, you should make sure that none of the predefined validators suites your needs. For two of the three validator types, there are already very powerful validation mechanisms at your hand: 1. the `regexp` matcher matches against an arbitrary regular expression, which means that you can handle a large class of string validations, 2. the `el` checker checks the converted input value using an arbitrary EL expression, so it might be a better solution to write a custom function (see *Calyxo Base* extensions) and use it in an EL expression. So, if these considerations didn't make you change your mind, don't hesitate to read on.

Technically, writing a new validator means to write a Java class which implements one of the *Calyxo Forms* validator interfaces (all of them can be found in the `de.odysseus.calyxo.forms` package). As you might expect:

- Every matcher has to implement the `Matcher` interface.
- Every converter has to implement the `Converter` interface.
- Every checker has to implement the `Checker` interface.

We'll have a closer look at those interfaces below, but first let's concentrate on the main ideas. Basically, when writing a new validator, you have to work out one or two methods, which define the behavior of your validator. Depending on the validator type, these are:

Validator Type	Method Name	Description
`Matcher`	`match(...)`	Processes its input string and returns a result string, usually the matched portion of the input string, which will be passed to the next validator in the validation sequence. If the matching fails, `null` is returned.
`Converter`	`parse(...)`	Parses its input string and returns an arbitrary result value, which will be passed to the first checker in the validation sequence, if there is one. If the parsing fails, a `ParseException` is thrown.
`Converter`	`format(...)`	Formats the given value from the data bean into its string representation. (This is not actually an act of validation, but its obviously a task, which has to be handled at the same place, since after parsing a formatted value the resulting value should not have changed.)
`Checker`	`check(...)`	Performs arbitrary tests on its input value and returns a `boolean` value of `true`, if it passed all tests successfully, and `false` otherwise.

After implementing your validator it has to be declared in the `<validators>` section of the *Calyxo Forms* configuration file, where it will be used. The declaration of a validator defines its name, its properties and the corresponding error message.

Example

The following code illustrates the implementation of a simple `CurrencyConverter`, which converts a string to an instance of `java.util.Currency` and vice versa. If the string to be parsed is empty, the string value of the property named `default` is parsed instead (whereas a `null` currency is formatted to a `null` string).

```
package my.validators;

import java.text.ParseException;
import java.util.Currency;
import de.odysseus.calyxo.forms.convert.SimpleConverter;

public class CurrencyConverter extends SimpleConverter {
  private String default;

  public CurrencyConverter() {
    super();
  }
  public String format(Object value) {
    if (value == null) {
      return null;
    }
    return ((Currency) value).toString();
  }
  public Object parse(String value) throws ParseException {
    if (value == null) {
      value = default;
    }
    try {
      return Currency.getInstance(value);
    } catch (IllegalArgumentException e) {
      throw new ParseException("Cannot parse: " + value, 0);
    }
  }
  public String getDefault() {
    return default;
  }
  public void setDefault(String value) {
    default = value;
  }
}
```

Here we use the convenience base class `de.odysseus.calyxo.forms.convert.SimpleConverter`, which implements the `Converter`

interface mentioned above. It declares two abstract methods, which have to be overridden with concrete versions: `public abstract Object parse(String value) throws ParseException` and `public abstract String format(Object value)`.

This converter is declared in a *Calyxo Forms* configuration file by adding a `<converter>` element to the `<validators>` section as shown below:

```
<validators>
  <converter id="currency" class="my.validators.CurrencyConverter">
    <property name="default" value="EUR"/>
    <message key="error.parse.currency" bundle="foo.msg">
      <arg name="field"/>
    </message>
  </converter>
</validators>
```

Now the converter is accessible in the `<forms>` section under the name `currency`, has a property called `default` (preset to `"EUR"`) and a predefined error message. The error message has to be added to the `msg.properties` file with an entry like `error.parse.currency=Field '{0}' must be a valid currency` and offers the argument named `field` to be filled with the actual field name when used:

```
<forms>
  <form name="currencyForm">
    <field property="preferredCurrency">
      <convert name="currency">
        <property name="default" value="USD"/>
        <message>
          <arg name="field" value="Preferred Currency"/>
        </message>
      </convert>
    </field>
  <form>
</forms>
```

In this case we override the default currency with the value `"USD"`, use the predefined error message and just set the field name to the value `"Preferred Currency"`, which causes the displayed error message to be `Field 'Preferred Currency' must be a valid currency`.

5. Integration

In order to use the *Calyxo Forms* component, it must be somehow integrated into the application's controller. This is achieved by the use of *plugins*.

- The component comes with a plugin for *Calyxo Control*.
- The *Calyxo Struts* component provides a similar plugin for Struts.

...

Furthermore, the plugins register the *Calyxo Forms* accessors.

5.1. Forms Plugin for Calyxo

The plugin provides a *filter* named `forms` for the *Calyxo Control* component. The `forms` filter can be used in `action` elements to validate incoming request data according to a form definition

The plugin is loaded in the module's controller configuration:

```
<calyxo-control-config version="0.9"
  xmlns="http://calyxo.odysseus.de/xml/ns/control">

  ...

  <plugin class="de.odysseus.calyxo.forms.control.FormsPlugin">
    <param name="config" value="/WEB-INF/calyxo-forms-config.xml"/>
  </plugin>

  ...

</calyxo-control-config>
```

The mandatory `config` parameter specifies the *Calyxo Forms* configuration file.

Forms Filter

To validate the inputs for a specific action you add a filter named `forms` to the action's definition:

```
<calyxo-control-config version="0.9"
  xmlns="http://calyxo.odysseus.de/xml/ns">

  ...

  <action path="/foo" source="page" target="page">
    <filter name="forms">
      <param name="form" value="fooForm"/>
      <param name="class" value="java.util.HashMap"/>
      <param name="attribute" value="fooData"/>
      <dispatch path="/WEB-INF/jsp/foo.jsp"/>
    </filter>
    ...
  </action>

  ...

</calyxo-control-config>
```

There are the following filter parameters:

`form`
Required. Determines the validation form, where the validation rules for this action are defined. The referred form has to be added to the *Calyxo Forms* configuration file (or to one of the files, which are imported by the configuration file).
`dispatch`
Optional. The name of the dispatch configuration used by the filter when form validation fails. If omitted, a nested anonymous `<dispatch>` element is required.
`class`
Optional. The fully qualified Java class to be used for the form data object, if none is found under the specified attribute name. The form data object is explained in more detail in the next section.
`attribute`
Optional. The name under which the form data is stored.
`scope`
Optional (requires the `attribute` parameter to be specified, too). The scope, where the form data is stored under the specified attribute name. Default value is `session`, alternatively you can choose `request`.
`commit`
Optional. A boolean value, which determines whether the validated form properties are automatically committed, i.e. stored in the form data, or not. (Default is `false`.)

Form Properties and Form Data

The validated input properties are accessible via an instance of `de.odysseus.calyxo.forms.FormProperties`, which is made available to your `execute(...)` method as described in the next section. The form properties offer a `getProperty(String name)` method to access an input property by name and a `commit()` method, which is used to populate the form data with the values from the form properties, if they passed all additional checks performed by your action. The form data object is an instance of the `de.odysseus.calyxo.forms.FormData` interface. This interface provides the property accessor methods `_getProperty(String name)` and `_setProperty(String name, Object value)`. When you specify your form data class (using the forms filter parameter named `class`), you have three possibilities:

- Your class may implement the `FormData` interface. In this case the form data object is simply an instance of the specified class, as you would expect.
- Your class may implement the `java.util.Map` interface. This causes the instance of the specified class to be wrapped in a `de.odysseus.calyxo.forms.misc.FormDataMap` object. This wrapper implements the `FormData` interface and gives access to the values of the contained map.
- Otherwise, we assume a bean and wrap the instance of the specified class in a `de.odysseus.calyxo.forms.misc.FormDataBean` object. As above, this wrapper implements the `FormData` interface and gives access to the properties of the contained bean.

In the latter two cases you can access the wrapped object (i.e., the map or the bean) using the `_getWrapped()` method provided by the `FormData` interface.

Actions

When writing your action class, you have two choices:

- You may subclass the convenience class `de.odysseus.calyxo.forms.control.FormsAction`, which itself is a subclass of `de.odysseus.calyxo.control.misc.AbstractAction`, (described in the *Calyxo Control* documentation). The `FormsAction` class adds the convenience method `getFormsSupport()`, which gives you an easy way to receive an appropriate instance of the utility class `de.odysseus.calyxo.forms.control.FormsSupport`, allowing you to handle the form properties, form data and form result in your action. Besides this, the `FormsAction` class replaces the abstract `execute(...)` method by an extended version, taking the form properties as an additional argument. It does now look like this:

  ```
  public abstract DispatchConfig execute(
    HttpServletRequest request,
    HttpServletResponse response,
    FormProperties formProperties) throws Exception;
  ```

  This way you have your form properties at hand when writing your `execute(...)` method, and thus all of the validated property values from your input form.
- If you prefer just to implement the `de.odysseus.calyxo.control.Action` interface (as described in the *Calyxo Control* documentation), you'll usually need to access your form properties, too. Then the above mentioned `FormsAction` class can give you a hint how to access it: just copy the code into your (root) action class and go ahead.

5.2. Forms Plugin for Struts

The forms plugin for Struts enables full use of the *Calyxo Forms* component with Struts. Plugin configuration details are covered in the *Calyxo Struts* documentation.

6. Project

6.1. History of Changes

Version 0.9.0 (2006/10/28)

developer: **cbe** ***context:*** **code** ***type:*** **update**
Minor refactoring to eliminate package cycle between `forms` and `forms.impl`.

Version 0.9.0-rc3 (2006/02/12)

developer: **cbe** ***context:*** **code** ***type:*** **update**
Renamed webapp `demo` to `test`.
developer: **cbe** ***context:*** **code** ***type:*** **update**

Eliminated dependencies on `commons-collections`.

developer: **cbe** ***context:*** **code** ***type:*** **update**

Fixed `ArrayIndexOutOfBoundsException` when trying to access array input with index beyond array size. Instead, behave as if no form data is available.

developer: **cbe** ***context:*** **code** ***type:*** **update**

Moved `impl.ValidationEngine` to `ValidatorBase`.

developer: **cbe** ***context:*** **code** ***type:*** **add**

Added convenience `checkbox` converter.

developer: **cbe** ***context:*** **code** ***type:*** **update**

The `boolean` converter no longer specifies `false` as its `default` property value (thus, it will map empty inputs to `null` unless `default` is set to another value within the `<convert>` element).

developer: **cbe** ***context:*** **code** ***type:*** **update**

Changed semantics of `<message>` element inside `<field>` to "factor out" defaults (attributes and `<arg>` elements) from `<message>` elements inside `<match>`, `<convert>` and `<check>` elements.

developer: **cbe** ***context:*** **code** ***type:*** **add**

Added `integer` converter. Left the equivalent old `int` converter for compatibility reasons. However, it may be removed in a future release.

Version 0.9.0-rc2 (2005/06/26)

developer: **all** ***context:*** **code** ***type:*** **fix**

The `RegexpMatcher` now always accepts an empty input.

developer: **all** ***context:*** **code** ***type:*** **add**

Added `ELChecker`.

developer: **all** ***context:*** **code** ***type:*** **update**

Renamed default value property of predefined converter `boolean` from `nullValue` to `default`.

developer: **cbe** ***context:*** **code** ***type:*** **fix**

Property message arguments of predefined validators may no longer be overridden (removed `name` attribute).

developer: **all** ***context:*** **docs** ***type:*** **add**

Improved documentation for predefined validators.

Version 0.9.0-rc1 (2005/03/07)

developer: **cbe** ***context:*** **code** ***type:*** **update**

Renamed filter parameter `source` to `dispatch`.

developer: **all** ***context:*** **code** ***type:*** **update**

Moved classes from `forms.selection` as well as group model interface and implementation to `forms.view`. Removed `SimpleListModel`.

developer: **all** ***context:*** **code** ***type:*** **update**

Eliminated `multiple` attribute from `<select>` tag; added `index` attribute to `<select>` tag.

developer: **all** ***context:*** **code** ***type:*** **update**

Removed selection-related methods from `ListModel`; added `selection` attribute to

<select> tag.
developer: **cbe** ***context:*** **code** ***type:*** **update**
Moved `ListModelAccessor` to package `misc`; Renamed list model converter property `name` to `attribute`; Renamed `calyxo-validators.*` to `calyxo-forms-validators.*`;
developer: **cbe** ***context:*** **code** ***type:*** **update**
Avoid `NumberFormat.getIntegerInstance()` in `Byte/Short/Integer/LongConverter`.
developer: **cbe** ***context:*** **code** ***type:*** **update**
Renamed list model accessor property `sort` to `values`.
developer: **cbe** ***context:*** **code** ***type:*** **add**
Introduced `FormContext` class and `FormComponent` interface. Allow tags to be used in JSP tagfiles (without form tag parent).
developer: **cbe** ***context:*** **code** ***type:*** **update**
Refactored `DefaultListModel` class.
developer: **cbe** ***context:*** **code** ***type:*** **fix**
`FormDataAccessor.get()` always returned `null`.
developer: **ost** ***context:*** **code** ***type:*** **add**
Introduced interface `FormProperties`.
developer: **cbe** ***context:*** **code** ***type:*** **add**
Added attribute 'final' to 'matcher', 'converter' and 'checker' elements
developer: **cbe** ***context:*** **code** ***type:*** **update**
Panels: renamed panel's `dispatch` attribute to `template`.
developer: **ost** ***context:*** **code** ***type:*** **update**
Renamed component `validation` to `forms`.

Version 0.9.0-b5 (2005/01/04)

developer: **cbe** ***context:*** **code** ***type:*** **add**
Added `validation.data` accessor to get form data.
developer: **ost** ***context:*** **code** ***type:*** **add**
Added time converter and bigInt converter, replaced bigDec converter. Added groupingUsed attribute to all number converters. Byte/short/int/long converters now use integerInstance of numberFormat.
developer: **ost** ***context:*** **code** ***type:*** **update**
Removed index attribute from checkbox, radio and radioitem tags. Removed var attribute from form tag (obsolete due to form data accessor).
developer: **ost** ***context:*** **code** ***type:*** **update**
Table demo replaces array demo. Enhanced types demo.
developer: **ost** ***context:*** **docs** ***type:*** **update**
Updates according to the changes listed above. Adapted links, source examples and tables to work with new version of forrester.

Version 0.9.0-b4 (2004/11/21)

developer: **cbe** ***context:*** **code** ***type:*** **update**
Updated plugin/filter/action code to `calyxo-control-0.9.0-b4`.
developer: **cbe** ***context:*** **code** ***type:*** **update**

The `test` attribute of the assert configuration element is now dynamic.

developer: **cbe** ***context:*** **code** ***type:*** **update**

All public methods in `ValidationSupport` that took a form name parameter now take an action path parameter instead.

Version 0.9.0-b3 (2004/10/20)

developer: **ost** ***context:*** **docs** ***type:*** **add**

Added documentation.

developer: **cbe** ***context:*** **code** ***type:*** **update**

Renamed `(error|assert)(style|class)` tag attribtes to `(error|assert)(Style|Class)`.

developer: **cbe** ***context:*** **code** ***type:*** **add**

Added `relax` attribute to `element` tag. This attribute takes a boolean EL expression. It is evaluated after validation of a form input failed. If the expression evaluates to `true`, producing an error message and emphasizing the HTML input tag will be suppressed.

developer: **cbe** ***context:*** **code** ***type:*** **add**

Added `ValidationSupport.removeFormData(HttpServletRequest, String)` method.

developer: **cbe** ***context:*** **code** ***type:*** **update**

Renamed attribute `type` to `class` in `matcher`, `converter` and `checker` configuration elements.

developer: **cbe** ***context:*** **code** ***type:*** **add**

Changed the use of list models in taglib: the `group` (and `select`) now may take a `listModel` attribute; added `listoption` tag; renamed `options` tag to `listoptions`.

Version 0.9.0-b2 (2004/07/04)

developer: **ost** ***context:*** **code** ***type:*** **fix**

Tag attributes and tag library descriptor

developer: **ost** ***context:*** **docs** ***type:*** **add**

Taglib reference

Version 0.9.0-b1 (2004/06/03)

developer: **cbe** ***context:*** **admin** ***type:*** **add**

First public release

6.2. Todo List

medium priority

- **[code]** Write more unit tests. >> Oliver
- **[docs]** Improve documentation. >> Oliver