

Calyxo MVC Web Application Framework

Table of contents

1 Calyxo Intro.....	5
1.1 General.....	5
1.1.1 Why another web application framework?.....	5
1.1.2 Download Calyxo.....	6
1.1.3 Calyxo installation instructions.....	7
1.2 Overview.....	8
1.2.1 Basics.....	8
1.2.2 Controller.....	11
1.2.3 View Management.....	13
1.2.4 Forms.....	17
1.3 Sample Application.....	19
1.3.1 Control.....	19
1.3.2 View.....	24
1.3.3 Final tasks.....	27
1.3.4 Deploying the application.....	29
1.3.5 Splitting the Application into Modules.....	30
1.3.6 Using Panels.....	32
1.3.7 Validating Forms.....	42
1.3.8 Using Struts as Controller.....	45
1.4 HOWTO's.....	45
1.4.1 Calyxo Eclipse Plugins.....	45
1.4.2 Deployment.....	46
1.4.3 Log4J Configuration.....	46
2 Calyxo Base.....	48
2.1 Concepts.....	49
2.1.1 Modules.....	50
2.1.2 Accessors.....	51
2.1.3 Internationalization.....	52
2.1.4 Configuration.....	54
2.2 Reference.....	61
2.2.1 Configuration.....	61

2.2.2	Accessors.....	70
2.2.3	Functions.....	74
2.2.4	Tag Library.....	75
2.3	Extension Points.....	78
2.3.1	Access API.....	78
2.3.2	EL Functions.....	81
2.4	Integration.....	82
3	Calyxo Control.....	83
3.1	Calyxo Control Concepts.....	83
3.1.1	Modules.....	85
3.1.2	Plugins.....	88
3.1.3	Actions.....	88
3.1.4	Exception Handlers.....	92
3.1.5	Dispatchers.....	93
3.1.6	Configuration.....	94
3.2	Reference.....	105
3.2.1	Configuration.....	105
3.2.2	Accessors.....	112
3.3	Miscellaneous.....	114
3.3.1	Adding Groovy Support.....	114
4	Calyxo Forms.....	116
4.1	Forms Concepts.....	116
4.1.1	Configuration.....	117
4.1.2	Validators.....	119
4.1.3	Forms.....	121
4.1.4	Fields.....	122
4.1.5	Inputs.....	124
4.1.6	Messages.....	125
4.1.7	Assertions.....	126
4.1.8	Actions.....	127
4.1.9	Tag Library.....	128
4.1.10	Selections.....	135
4.1.11	l18n.....	140
4.2	Reference.....	140

4.2.1 Configuration.....	140
4.2.2 Accessors.....	151
4.2.3 Tag Library.....	152
4.2.4 Predefined Validators.....	167
4.3 Extension Points.....	172
4.3.1 Adding your own validators.....	172
4.4 Integration.....	175
4.4.1 Forms Plugin for Calyxo.....	175
4.4.2 Forms Plugin for Struts.....	177
5 Calyxo Panels.....	178
5.1 Panels Concepts.....	179
5.1.1 Configuration.....	180
5.1.2 Panels.....	181
5.1.3 Parameters.....	183
5.1.4 Lists.....	184
5.1.5 Inheritance.....	185
5.1.6 I18n.....	186
5.2 Reference.....	187
5.2.1 Configuration.....	187
5.2.2 Accessors.....	191
5.2.3 Tag Library.....	192
5.3 Integration.....	194
5.3.1 Panels Plugin for Calyxo Control.....	194
5.3.2 Panels Plugin for Struts.....	195
6 Calyxo Struts.....	196
6.1 Calyxo Struts Plugins.....	196
6.1.1 Base plugin for Struts.....	196
6.1.2 Panels plugin for Struts.....	198
6.1.3 Forms plugin for Struts.....	200

1. Calyxo Intro

The *Calyxo* framework encourages in MVC model 2 based web application development. It offers support for true modular applications, i18n, a flexible view manager, a powerful validation engine, and more! *Calyxo* is entirely written in Java and builds on the latest Servlet and JSP technologies.

Calyxo is developed by [Odysseus](#), a software company located in Frankfurt, Germany. *Calyxo* is made available under the [Apache License 2.0](#). The project is hosted at SourceForge.net. Consult the [Calyxo Project Page](#) for further information.

Components

The *Calyxo* project divides into several components

- **Calyxo Base** – this component collects some of the basic, reusable classes used throughout all the other components. It introduces basic concepts like modules, i18n, accessors and so on...
- **Calyxo Control** – this component implements the *Calyxo* controller. *Calyxo* uses an approach similar to Struts here, so Struts users should feel familiar with *Calyxo* from the start. *Calyxo Control* may be considered as the "main part" of the *Calyxo* platform.
- **Calyxo Forms** – this component allows the definition of forms, along with all their validation steps and lets you map inputs to properties of a data object. Complex validations, dependencies between inputs and mapping an input to a set of properties are supported. The *Calyxo Forms* JSP tag library provides replacements for HTML elements related to forms.
- **Calyxo Panels** – this component allows the definition of pages in a container/component manner. Pages are dynamically composed from a tree of page fragments. Panel definitions may be nested to arbitrary depth and may use inheritance.
- **Calyxo Struts** – this component focuses on integrating *Calyxo* parts into the very popular Apache Struts framework. The *Calyxo Panels* and *Calyxo Forms* components may be used with Struts to replace Struts Tiles and the Struts Validator.

Requirements

Calyxo requires Java 1.4.2 or later as well as a servlet container supporting the Servlet 2.4 / JSP 2.0 APIs (eg. Tomcat 5).

1.1. General

1.1.1. Why another web application framework?

Since the late 90's, a whole bunch of web application frameworks based on servlet technology appeared (and some also disappeared) on the horizon.

The one that seems to come close to many people's needs is the popular [Struts](#) framework. We have been using Struts for over three years now, and - all in all - we liked it. Struts spread the MVC Model 2 approach all over the world and introduced some very good ideas. Indeed, Struts influenced *Calyxo* in many areas.

So, where's the need for yet another framework? We felt, that basically we wanted to use the Struts controller, but wanted to get rid of the things, that gathered around it. Neither the tiles mechanism nor the validator satisfied our needs. Furthermore, since the availability of JSTL 1.1, 90 percent of Struts' tag libraries are obsolete.

The greatest needs came up in the user input validation area. So, we started to develop a validation component as a Struts plugin. Another issue was view-management (define and parametrize your view templates - e.g. JSP fragments - in a centralized file). To address this, we developed another component as a Struts plugin.

However, the Struts controller's extension capabilities are quite limited (you can do it, but you can't do it well). On the other hand, the controller part is just a couple of classes, so that would not justify the use of Struts. Went so far, we decided to provide our own controller, which takes the best of Struts' controller and adds some nifty features we all desired.

The result is *Calyxo*, a complete web application framework, which not only can speed up and simplify your development, but can also improve the quality of your applications.

1.1.2. Download Calyxo

You can choose between a source distribution or binary distribution. We strongly recommend that you start with the binary distribution, since it contains all the libraries you need to start right away.

Source code

Even if you want the *Calyxo* sources, you can choose the binary distribution. It contains the full source code as zip archives.

Current Release

You can download the current *Calyxo* release via the [Calyxo Files](#) page at Sourceforge.

PDF Manual

The complete *Calyxo* manual is available in PDF format as a separate download via the [Calyxo Files](#) page at Sourceforge. As an alternative, you may access the latest version [here](#) (~1M).

Eclipse Plugins

The *Calyxo* Eclipse are not part of the *Calyxo* core. They are separately available at *Calyxo*'s

[Eclipse Update Site](#) and can be installed using the Eclipse update manager. Refer to the [Eclipse Setup HOWTO](#) for further information.

1.1.3. Calyxo installation instructions

System requirements

Calyxo requires

- a Java runtime environment, version 1.4.2 or later. We recommend to get a recent [J2SE](#) from Sun.
- a servlet container supporting Servlet API 2.4 and JSP API 2.0. [Tomcat 5](#) supports these and may be a good choice for development. However, since [J2EE 1.4](#) covers all the above, you can run Calyxo applications in any J2EE 1.4-compliant application server.

Now, if you installed all the prerequisites, you should [download Calyxo](#).

Installing the binary distribution

The binary distribution contains all you need to develop and run applications based on Calyxo. Unpack the archive to some location of your choice and - you're done.

Eclipse users

If you're using [Eclipse](#), you may be interested in our suggestions on how to [setup Eclipse](#) to develop Calyxo applications.

The Calyxo binary distribution directory tree should look similar to this:

```
calyxo-0.9.0/  
  calyxo-base/  
  calyxo-control/  
  calyxo-demo/  
  calyxo-forms/  
  calyxo-panels/  
  calyxo-struts/  
  ...  
  build.xml  
  ...
```

As you can see, there's one subdirectory per Calyxo component. Some of these components contain sample web applications.

Building the Demo Application

The calyxo-demo subdirectory contains the Calyxo demo. To build it, simply run `ant` in the Calyxo binary distribution directory.

That's it! You can find the `calyxo-demo-0.9.0.war` file in the `calyxo-demo` subdirectory. The deployment process of web applications depends on the application server you use.

Tomcat 5 users

Simply copy the `calyxo-demo-0.9.0.war` file to the `webapps` directory below your tomcat home directory.

Building Calyxo from source

If you want to build *Calyxo* from source, you should have downloaded and unpacked the *Calyxo* source distribution.

Warning

The source distribution does not contain any third party libraries required by *Calyxo*. You will have to get them separately to be able to build *Calyxo*.

- Change to the *Calyxo* source distribution directory.
- Copy `build.properties.sample` to `build.properties` and edit it according to your system environment.
- Some of the various component subdirectories may also contain `build.properties.sample` files. If so, process these as described above.
- In the *Calyxo* source distribution directory, run `ant`.

1.2. Overview

In an MVC Model 2 application, Servlets and JSPs are used together, but have completely different roles.

- Servlets are used as part of the controller. An incoming request is handled by executing a particular piece of code, called an *action*, which in turn interacts with the application's model.
- JSPs are used as part of the view. This means, after one or more actions have been executed, a request is dispatched to a JSP page. The view is responsible for displaying data provided by the actions. Doing this may bring up formatting and i18n issues. However, in no circumstances, it performs business logic.

The promise of the MVC Model 2 approach is to increase reusability, scalability and maintainability while reducing complexity. In practice, it is very hard to build a Model 2 application from scratch. This is where the need for a framework like *Calyxo* comes into play.

Let us give you a brief introduction to *Calyxo*'s components.

1.2.1. Basics

Calyxo itself follows a component based approach. This means, parts of *Calyxo* may be used alone. For example, the *Calyxo Panels* and *Calyxo Forms* components may be used with

Struts. However, all *Calyxo* components share some common concepts and properties, implemented in the *Calyxo Base* component.

Let's examine some of these next.

1.2.1.1. Modules

Calyxo applications are composed of *modules*. Modules are independent units, you may think of them as subapplications. A module is a container for *actions*. An action will be invoked (or executed), when it is selected by a request.

A module is represented by its *context*. The module context provides access to module properties and services, like

- the module name
- the surrounding servlet context
- module initialization parameters
- module scope attributes
- the context relative path for a given action (module relative path).

As you can see, a module context provides its own attribute scope, just like the servlet context does. Only, it is private to that module. *Calyxo* stores all its configuration information inside module scopes to prevent name clashes between modules.

In a real application, an incoming request has to be mapped to its corresponding module. That is, a context relative path gets decomposed into a module part (identifying the module) and an action path (relative to the module). As mentioned above, a module context provides the inverse mapping, answering the question: how do I access a particular action within the module from outside?

Calyxo provides access to module contexts through *accessors*, that may be used in JSTL expressions, for example

```
${calyxo.base.module.name}
${calyxo.base.module.attribute['foo']}
${calyxo.base.module.forName['clients'].path['/list']}
```

The last example may be used to build URLs pointing to an action in some specific module. However, *Calyxo* also provides a custom JSP tag, that clones HTML's <a> tag, but replaces the href attribute with module and action attributes.

1.2.1.2. Internationalization

Calyxo supports i18n from the ground up. At the very basic, *Calyxo* provides ways to resolve resources in a locale dependent manner. A resource is identified by a bundle name and a resource key.

A resource may be a template, that expects arguments to be expanded to a *message*.

The default mechanism for resources is to use Java's ResourceBundle class. Messages are

created by using Java's `MessageFormat` class, by default. However, your application may provide its own or customized mechanism.

Calyxo supports i18n of views through accessors in a way, that makes localizing content very easy. Just to let you taste: in your JSPs you can use JSTL expressions like

```
${calyxo.base.i18n.bundle['strings'].resource['user_id']}
${calyxo.base.i18n.bundle['strings'].message['required']['user_id']}
```

Thus, you do not need any custom tag libraries to localize your application.

The *Calyxo* components have been developed with i18n issues in mind. For example, the *Calyxo Panels* component allows to define locale dependent views. The *Calyxo Forms* component supports locale dependent form definitions.

1.2.1.3. Configuration

Each module takes its own set of configuration files. Within a module, each *Calyxo* component is configured through individual configuration files.

Generally, all *Calyxo* configuration files are in XML format and share the ability to

- import another configuration file: the imported configuration is *merged* into the importing configuration.

```
<base:import file="../../calyxo-control-config-shared.xml"/>
```

This feature greatly supports sharing of common configuration parts between modules.

- define, initialize and store Java objects as local variables or as module (or application) scope attributes:

```
<base:set var="content" value="/WEB-INF/content"/>
```

or even

```
<base:set var="mybean" scope="module">
  <base:object class="...MyBean">
    <base:property name="foo" value="bar"/>
  </base:object>
</base:set>
```

Simple, but *very* useful...

- use JSP EL expressions in so called *dynamic* attributes to reference local variables and attributes in module- or application scope.

```
<foo value="${mybean}"/>
```

- declare functions to be used in dynamic attributes: you can provide you own function libraries.

```
<base:functions prefix="bar" class="...MyFunctions"/>
```

1.2.1.4. Accessors

Calyxo provides so-called *accessors*, to allow easy access to *Calyxo*-related information from within your views. Accessors form a tree of Java Beans and `java.util.Maps`, that may be queried through JSTL expressions, anywhere in a page.

The tree of accessors is instantiated and installed into request scope using the `<base:access>` tag, like in

```
<base:access var="calyxo"/>
```

By convention, we use `calyxo` to denote the root.

As an example, to take property `foo` from the bean at attribute `mybean` in module scope, you would use an expressions like

```
${calyxo.base.module.attribute['mybean'].foo}
```

The various *Calyxo* components contribute accessors providing information relevant to that component. However, *Calyxo* even lets you extend the tree by implementing your own accessors.

1.2.2. Controller

The controller lies at the heart of an MVC Model 2 application. In a *Calyxo* application, each module is associated with its own servlet. For an incoming request, a *Calyxo* module is selected by the servlet container, according to the url mappings, we associated with our modules. We call the selected module the *current* module.

The current module then selects an *action* and executes it. We say, "*the action is invoked by the request*". An action's execution results in information on how to *dispatch* the request. The module then *dispatches* to another action (optionally within another module) or to an application resource (for example, a JSP page). Technically spoken, the module uses a *dispatcher*, which usually performs a request *forward* (or *include*).

It is important to understand, that the current module exists only during requests handled by a *Calyxo* module. If you, for example, point your browser directly to a JSP page, there's no current module! If, on the other hand, a *Calyxo* action dispatches to that page, the action's module will still be the current module within the page.

Implementing actions is your part. They are the controller's end, that invoke your application specific business logic.

The *Calyxo* controller provides several extension points. *Plugins* are used to load extensions. Plugins may define *filters*, which are used to extend an action by a filter chain. Or, they may define *dispatchers* to customize request dispatching to particular targets. Please, consult the [Calyxo Control documentation](#) for more on this.

1.2.2.1. Configuration elements

Like all *Calyxo* components, the controller is configured by one or more XML configuration files. The most essential elements are

- *action* elements: an action element defines how to handle a request for some module-relative path. An action may specify an *action class* to be invoked. For example,

```
<action path="/show" class="...ShowAction">...</action>
```

will invoke `...ShowAction`, when the module selects action `/show`.

- *dispatch* elements: a dispatch element defines a branch of control under some name. You can branch to another action or an application resource. For example,

```
<dispatch name="foo" module="clients" action="/show">
```

```
  ...
```

```
</dispatch>
```

defines, that dispatching to `foo` will result in executing action `/show` in module `clients`. The module attribute is optional and needed only for module switches. On the other hand,

```
<dispatch name="bar" path="/WEB-INF/jsp/show.jspx">
```

```
  ...
```

```
</dispatch>
```

defines, that dispatching to `bar` will result in displaying JSP page `/WEB-INF/jsp/show.jspx`. Dispatch elements may appear inside action elements or as *global* dispatches, visible to all actions.

- *filter* elements: actually, an action is embedded into a chain of action filters. Filters may intercept before and after the action has been executed. Typical candidates for filters are: cancelled forms, input validation, role based security, ... As an example,

```
<filter name="foo">...</filter>
```

inside an action element will add the `foo` filter to the action filter chain.

- *param* elements: all elements mentioned so far may be parametrized with param elements. A parameter simply has a name and a string value. For example,

```
<param name="foo" value="bar"/>
```

inside an action, dispatch or filter element adds a parameter with name `"foo"` and value `"bar"` to the the configuration element. Dispatch parameters are added as parameters to the forwarded request.

The controller configuration also provides other elements, that we won't cover here. For example, you may define exception handlers and plugin extensions.

1.2.2.2. Action classes

Calyxo provides the `de.odysseus.calyxo.control.Action` interface. All action implementations must implement this interface. Actions are instantiated and initialized by the controller during start up.

The Action interface defines two methods:

- After instantiation, the controller gives the action the opportunity to do some initialization.

An object reflecting the action configuration element and the module context are passed in.

```
public void init(
    ActionConfig config,
    ModuleContext context) throws Exception;
```

- The execute method will be called to "invoke" the action, passing the request and response objects as parameters. It returns an object reflecting a dispatch configuration element.

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

For your convenience, there is also an abstract base action implementation, namely `de.odysseus.calyxo.control.AbstractAction`. This class provides methods to serve the module context, action configuration, as well as a "message support" to save error, warning and info messages.

1.2.3. View Management

Many aspects of a web application's presentation layer are common to all views. For example, pages may be composed of header, footer, menu and content areas. To keep maintainability and consistency, reuse of view components is an important issue.

To achieve reuse, we split our views into *templates*, which are combined to pages at runtime. A template may include other templates and may be included by other templates. Thus, an actual page may be seen as a template tree.

Next, we need a flexible mechanism to specify *how* each page is to be composed of templates. In *Calyxo* terminology, this is done using *panels*:

- a panel is associated with a template
- panels may be nested to arbitrary depth
- a panel may pass parameters to its associated template

After you defined some panels, *Calyxo* can use them to compose and render the corresponding templates.

1.2.3.1. Panel definitions

Panels are defined per module in an XML configuration file. As stated earlier, a panel definition corresponds to a template. The template includes other template corresponding to the subpanels in the definition. In a JSP environment, templates are just JSPs.

Let's make this clear using an example. The panels configuration file may contain a panel definition like this:

```
<panel name="/base.page" template="/WEB-INF/jsp/layout/page.jsp">
  <panel name="header" template="/WEB-INF/jsp/layout/header.jsp"/>
```

```

<panel name="menu" template="/WEB-INF/jsp/layout/menu.jsp"/>
<panel name="content"/>
<panel name="footer" template="/WEB-INF/jsp/layout/footer.jsp"/>
</panel>

```

- The /base.page panel defines the subpanels header, menu, content and footer. The template attributes denote the corresponding JSP templates.
- The name attribute is mandatory for panels. By convention, we assign path-like names to toplevel panels and identifier-like names to subpanels.
- The content subpanel does not specify a corresponding template, making the definition *abstract*.

The /WEB-INF/jsp/layout/page.jsp template may include a subpanel using the <panel> tag from the custom tag library provided by the *Calyxo Panels* component like this:

```

<jsp:root version="2.0"
  xmlns:panels="http://calyx.odyseus.de/jsp/panels"
  xmlns:jsp="http://java.sun.com/JSP/Page">
  ...
  <panels:panel name="header"/>
  ...
  <panels:panel name="menu"/>
  ...
  <panels:panel name="content"/>
  ...
  <panels:panel name="footer"/>
  ...
</jsp:root>

```

1.2.3.2. Parameters

You may define static parameters within your panel definitions. These parameters are accessible from your panel fragments. Define a parameter like this:

```

<panel name="...">
  ...
  <param name="foo" value="bar"/>
  ...
</panel>

```

Now, in the corresponding template, you may access the panels' parameters from a JSTL EL expression using the `calyx.panels.param` accessor:

```

<jsp:root version="2.0"
  xmlns:panels="http://calyx.odyseus.de/jsp/panels"
  xmlns:jsp="http://java.sun.com/JSP/Page">
  ...
  <!-- access parameter via EL -->
  <someTag someattr="{calyx.panels.param['foo']}">...</someTag>
  ...

```

```
</jsp:root>
```

1.2.3.3. Inheritance

Since our previous definition is abstract, we need to *extend* it to get a *concrete* panel definition:

```
<panel name="/base.page" template="/WEB-INF/jsp/layout/page.jsp">
  <panel name="header" template="/WEB-INF/jsp/layout/header.jsp"/>
  <panel name="menu" template="/WEB-INF/jsp/layout/menu.jsp"/>
  <panel name="content"/>
  <panel name="footer" template="/WEB-INF/jsp/layout/footer.jsp"/>
</panel>

<panel name="/derived.page" super="/base.page">
  <panel name="content" template="/WEB-INF/jsp/content/foo.jsp"/>
</panel>
```

The `super` attribute is used to specify a toplevel panel as a base panel definition. In the above example, the `/derived.page` panel associates a template with the content subpanel, making it concrete.

Let us examine a slightly more complex example:

```
<panel name="/base">
  <panel name="nested">
    <param name="param1" value="p1"/>
    <param name="param2" value="p2"/>
    <panel name="nested1nested"/>
  </param>
</panel>

<panel name="/derived" super="/base" template="/WEB-INF/derived.jsp">
  <panel name="nested" template="/WEB-INF/nested1.jsp">
    <param name="param1" value="p1"/>
    <param name="param2" value="override p2"/>
    <param name="param3" value="add p3"/>
  </param>
  <panel name="nested2" template="/WEB-INF/nested2.jsp"/>
</panel>

<panel name="/concrete" template="/WEB-INF/derived2.jsp">
  <panel name="foo" super="/derived">
    <panel name="nested">
      <panel name="nested1nested" template="/WEB-INF/bar.jsp"/>
      <param name="param3" value="override p3"/>
    </panel>
  </panel>
</panel>
```

Don't run away! It looks harder than it is... Let's work out what happens here:

- Toplevel panel `/base` defines an abstract subpanel `nested`. It is abstract, because it does not specify a `template` attribute and also, because it has an abstract subpanel `nested1nested`.
- Panel `/derived` extends `/base` and assigns a `template` value to `nested`, assigns a value to `nested's param1`, overrides the `param2` parameter and adds the `param3` parameter. Finally, it adds `nested2`, another nested panel. However, `/derived` is still abstract, because it inherits the abstract `nested1nested` panel.
- The `/concrete` panel definition takes a `foo` subpanel, which is derived from `/derived`. The `foo` panel overrides `param3`. Since it assigns a `template` attribute to `nested1nested`, `foo` is concrete and thus `/concrete` is concrete.

Got it?

1.2.3.4. Lists

Besides subpanels and parameters, a panel may also contain lists. A list contains a sequence of items containing panels, parameters and - you guessed it - lists. The *Calyxo Panels* custom tag library provides a tag that allows templates to iterate over a list in its corresponding definition. Lists may be used to define menu structures, a sequence of panels to be layout in a specific way, and more.

As an example, consider the following panel definition:

```
<panel name="/blurb" template="/jsp/column.jsp">
  <list name="items">
    <item>
      <param name="head" value="Item 1"/>
      <panel name="body" template="/jsp/blurb1.jsp"/>
    </item>
    <item>
      <param name="head" value="Item 2"/>
      <panel name="body" template="/jsp/blurb2.jsp"/>
    </item>
    <item>
      <param name="head" value="Item 3"/>
      <panel name="body" template="/jsp/blurb3.jsp"/>
    </item>
  </list>
</panel>
```

The `/blurb` panel defines the `items` list, containing three items, each containing a parameter named `head` and a panel named `body`.

The layout template `/jsp/column.jsp` iterates over the list:

```
<jsp:root version="2.0"
  xmlns:panels="http://calyx.odyseus.de/jsp/panels"
  xmlns:jsp="http://java.sun.com/JSP/Page">
```

```

<panels:list name="items">
  <h4>${calyxo.panels.param.head}</h4>
  <p>
    <panels:panel name="body"/>
  </p>
</panels:list>
</jsp:root>

```

During iteration, the current item behaves as if its contained elements (head and body) were moved up to the panel containing the list. Thus, when rendering `/blurb`, the above template creates paragraphs containing contents of `/jsp/blurb1.jsp` with heading "Item 1", `/jsp/blurb2.jsp` with heading "Item 2" and `/jsp/blurb3.jsp` with heading "Item 3".

1.2.3.5. I18n

The *Calyxo Panels* component supports locale-dependent variants of panel definitions. That is, the selection of panels may depend on the user's locale. The details on this are beyond the scope of this overview. Consult the [Calyxo Panels documentation](#) for further information.

1.2.4. Forms

Validation and presentation of user inputs is essential for a good web user interface. The *Calyxo Forms* component offers a rich set of important features in this area:

- The conversion of validated input data into appropriate data types, ready for subsequent use.
- The generation of pregnant error messages for non-validated inputs.
- Marking of those input fields, that contain non-validated inputs
- I18n-aware validation, data formatting and error messages.
- Formatting of validated inputs according to common standards.
- Flexible validation rules for single fields as well as conditions involving multiple fields.

Form validation is performed in two phases. First, every input is undergoing a sequence of *field validations*. Then, *assertions* can be used to test for complex conditions. Assert expressions use the JSP Expression Language (EL) and may reference any form field values.

A form validation succeeds, if all form inputs could be validated and no assertions failed. After a form has been validated successfully, a *form data* object, associated with the form, is populated with the validated data. The form data object may be accessed in subsequent request processing, eg in an action.

1.2.4.1. Fields

A *field* corresponds to an input parameter and a form data property. It is associated with a sequence of validations. The field is said to be *valid* if all validations in the sequence succeed. We say, the input *is mapped by that field*, meaning that the form data property of that field will be set to the field's value.

Field validations fall into three categories: *matchers*, *converters* and *checkers*. A field's validation sequence consists of zero or more matchers, followed by zero or one converter, followed by zero or more checkers.

1. A *matcher* takes a string as input and produces a string as a result. As the name suggests, the result string usually should be a substring of the input string. However, this is not required. Typical matchers strip whitespace or match against a regular expression. Matchers are connected as a pipeline: the first matcher takes the original form parameter as input. A matcher's output is taken as input by its successor.
2. The *converter* takes a string as input and produces an object as result. It *parses* a string to produce an object of some type, eg a number or date. The converter takes the result of the matcher pipeline as its input.
3. A *checker* takes an object and performs some tests on it. As its result, it returns a boolean to indicate success or failure. Typical checkers may perform range tests on numbers or dates, length tests on strings, and so on. A checker takes the converted value as input.

There's a one-to-one correspondence between fields and form data properties. Usually, a field also corresponds to one input parameter.

Advanced

However, there may be more than one field for an input parameter. In this case, the input is mapped by the first valid field. The form data properties corresponding to the other fields for that input will be set to null. For example, this feature may be used to have an input that may contain a date or a number.

When a field validation fails, *Calyxo* marks the corresponding input to enable visual feedback to the user. For example, invalid input fields may be colored red.

1.2.4.2. Assertions

Often, there's a need to check *inter-field dependencies*. For example, one may want to make sure, that the two given passwords are the same, two dates are within one week, or that some field is required if another field is non-empty.

To address this, *Calyxo*'s form validation process may be configured to perform *assertions*. An assert expression is an arbitrary complex JSP EL expression, which may reference request parameters, form inputs and form data properties.

- Request parameters are referenced by the implicit object `param`, as in `param.foo`.
- Form inputs are referenced by the implicit object `input`, as in `input.foo`.
- Validated form data properties are referenced by the implicit object `property`, as in `property.bar`.

Note

Usually, you may assume form input `foo` to be the same as request parameter `foo`. However, there may be request parameters that are not specified as form inputs.

When an assertion fails, *Calyxo* marks the involved inputs to enable visual feedback to the user. For example, input fields involved in a failed assertion may be colored orange.

1.2.4.3. i18n

As we have seen, field validation involves matchers, a converter and checkers. Since we want to support locale-dependent field validation, matchers, converters and checkers are *localized*. Converters are also used to *format* objects back to some normalized, locale-dependent, display strings.

Validation error messages are localized with *Calyxo*'s i18n support, as usual.

The *Calyxo Forms* component supports locale-dependent variants of form definitions. That is, the selection of the form definition used for validation may depend on the user's locale. Please, consult the [Calyxo Forms documentation](#) for further information.

1.3. Sample Application

To ensure your environment is set up correctly, you should have built and run the demo application successfully, before you start with this tutorial. Refer to the [installation instructions](#) for more on this.

Our application will come up with a login dialog, where the user is prompted to type in his user id and password. When the user submits the form, an *action* will verify the user data and

- display a welcome page on success
- redisplay the input form with an appropriate error message else

Not too complex, but sufficient to show the principles.

First, we'll create a single-module application, using just the *Calyxo* controller. Then, we'll divide it into modules. Finally, we'll add view management and validation capabilities.

Before we start, we should give our application a home. Anywhere you like, create a directory structure like this:

```
login-sample/  
  WEB-INF/  
    classes/  
    jsp/  
    lib/
```

1.3.1. Control

In the simplest case, an application consists of only one module. We'll start with a single module and divide it into two modules later.

To bring our application's controller to life we need to

- provide a configuration file per module to define our actions,

- implement our action classes,
- define a servlet and servlet mapping per module in the web application deployment descriptor.

1.3.1.1. Configuring the controller

We'll place our controller configuration into `/WEB-INF/calyxo-control-config.xml`. It looks like this:

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

  <actions>

    <!-- The index action just forwards to our login page -->
    <action path="/index">
      <dispatch path="/WEB-INF/jsp/login.jspx"/>
    </action>

    <!-- Login action -->
    <action path="/login" class="de.odysseus.calyxo.sample.login.LoginAction">
      <dispatch name="success" path="/WEB-INF/jsp/welcome.jspx"/>
      <dispatch name="input" path="/WEB-INF/jsp/login.jspx"/>
    </action>

    <!-- Logout action -->
    <action path="/logout"
      class="de.odysseus.calyxo.sample.login.LogoutAction">
      <dispatch name="success" path="/WEB-INF/jsp/goodbye.jspx"/>
    </action>

  </actions>

</calyxo-control-config>
```

DTD lookup

Many XML editors allow to associate root element names with DTDs. If your editor supports this, you may want to associate `calyxo-control-config` with `CALYXO_HOME/calyxo-control/conf/share/calyxo-control-config.dtd`. Alternatively, you should adjust the DTD system path or simply copy the DTD file to your `/WEB-INF` directory.

Let's explore the elements and attributes used above:

- You must declare the `xmlns="http://calyxo.odysseus.de/xml/ns/control"` and `version="0.9"` attributes in the root element.
- The `actions` element contains a sequence of action elements, which define the entry points of the module.
- An action has a mandatory `path` attribute. The action path is the action's module relative address. It must start with a slash (/).

- An action may contain dispatch elements. A dispatch has a name attribute. By omitting this attribute, the action's default dispatch is defined. A dispatch may branch to another action by specifying the action attribute or to an application resource by specifying the path attribute.
- If an action has a class attribute, its value is taken to be an action class name (subclass of `de.odysseus.calyxo.control.Action`), whose `execute()` method will be called to invoke that action.
- If an action's class attribute is omitted, the controller will insert a default action, which dispatches according to the default dispatch element (the one without a name attribute).

Our sample configuration only uses a small subset of the available elements. Beyond what you have seen so far, you can

- define *exception handlers*
- define global *dispatches*
- pass *parameters* or *parameter sets* to actions
- declare *plugins* to be loaded by the module
- use your own *dispatchers*
- use *filters* to build action chains

For these advanced features of the controller, please refer to the [Calyxo Control](#) component documentation.

1.3.1.2. Implementing the actions

Due to our configuration, we need to implement an action to login and another action to logout. Instead of directly implementing the Action interface, we inherit from the convenience `AbstractAction` base implementation.

Login action

```
package de.odysseus.calyxo.sample.login;

import java.util.HashMap;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import de.odysseus.calyxo.base.Message;
import de.odysseus.calyxo.control.misc.AbstractAction;
import de.odysseus.calyxo.control.conf.DispatchConfig;

public class LoginAction extends AbstractAction {

    private HashMap users = new HashMap();

    /**
     * Initialize our "user database" map
```

```

    */
public void init() {
    users.put("jeff", "hacker");
    users.put("fred", "tester");
    users.put("joe", "manager");
}

/**
 * Process login request.
 * The specified user id and password is verified against
 * our map.
 * On success, the user name is placed into session scope.
 */
public DispatchConfig execute(
    HttpServletRequest request,
    HttpServletResponse response) throws Exception {

    String user = request.getParameter("user");
    if (!users.containsKey(user)) {
        Message.Arg arg = new Message.ValueArg(user);
        Message message = new Message("messages", "login.user.unknown", arg);
        getMessageSupport().addError(request, "user", message);
        return getActionConfig().findDispatchConfig("input");
    }
    if (!users.get(user).equals(request.getParameter("password"))) {
        Message message = new Message("messages", "login.failed");
        getMessageSupport().addError(request, message);
        return getActionConfig().findDispatchConfig("input");
    }
    request.getSession().setAttribute("user", user);
    return getActionConfig().findDispatchConfig("success");
}
}

```

As you can see, the login action does not validate inputs. Later, when we use the *Calyxo Forms* component, we'll be able to perform the necessary validations without changing any code here.

But there's another point of interest: the action produces localized messages and saves them using the `addError()` methods. When implementing our view, we should not forget to show up these messages somehow.

Logout action

```

package de.odysseus.calyxo.sample.login;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

```

```

import de.odysseus.calyxo.control.misc.AbstractAction;
import de.odysseus.calyxo.control.conf.DispatchConfig;

public class LogoutAction extends AbstractAction {
    /**
     * Process logout request.
     * Remove user name from session scope.
     */
    public DispatchConfig execute(
        HttpServletRequest request,
        HttpServletResponse response) throws Exception {

        request.getSession().removeAttribute("user");
        return getActionConfig().findDispatchConfig("success");
    }
}

```

Trivial, isn't it?

1.3.1.3. Activating the module

We're almost done with the controller part of our sample application. Now, let's make our module known to the servlet container. This is done in the web application's deployment descriptor, which has to be made available as `/WEB-INF/web.xml`.

To specify our *login-sample* module, we have to do the following:

- define a servlet for our module
- pass it the configuration file as an initialization parameter
- map an url pattern to the servlet

Here's our `web.xml`:

```

<web-app xmlns="http://java.sun.com/xml/ns/j2ee" version="2.4">

    <!-- Module Servlet -->
    <servlet>
        <servlet-name>login-sample</servlet-name>
        <servlet-class>de.odysseus.calyxo.control.ModuleServlet</servlet-class>
        <init-param>
            <param-name>config</param-name>
            <param-value>/WEB-INF/calyxo-control-config.xml</param-value>
        </init-param>
        <load-on-startup/>
    </servlet>

    <!-- Module Mapping -->
    <servlet-mapping>
        <servlet-name>login-sample</servlet-name>
        <url-pattern>*.do</url-pattern>
    </servlet-mapping>

```

```

<!-- Welcome File -->
<welcome-file-list>
  <welcome-file>index.jsp</welcome-file>
</welcome-file-list>

</web-app>

```

The url mapping we chose to address our module is a so called *extension mapping*: all context relative paths ending with `.do` select our module. Alternatively, we could have used a *prefix mapping* like `/main/*`: all context relative paths starting with `/main/` select our module.

That's it!

1.3.2. View

So far, our sample application is ready to serve requests. The missing link is the user interface.

JSP pages

Currently, *Calyxo* supports JSP 2.0 as a view technology. So, in this section, we'll create the necessary JSP pages.

Instead of 50.000 custom JSP tags, *Calyxo* provides so-called *accessors*, a hierarchy of Java objects, which can be used in JSTL EL expressions from within your JSP pages to access *Calyxo*-related information.

JSP syntax

You may already have noticed, that there are two alternative JSP syntaxes: the traditional, non-XML syntax, and the newer, XML-based syntax. Throughout this tutorial, we'll use the XML-based syntax. Just in case you wonder about that: **it is up to you, which syntax you use in your applications.**

i18n

The second issue we have to deal with is i18n. At least, we have to provide message resources. Optionally, we can externalize all localized content from our pages.

1.3.2.1. Creating the JSP files

Our controller configuration references three JSP files: `/WEB-INF/jsp/login.jsp`, `/WEB-INF/jsp/welcome.jsp` and `/WEB-INF/jsp/goodbye.jsp`.

Login page

Our login JSP page contains the form used to submit the user's id and password. Here's the JSP document:

```
<jsp:root
  xmlns:jsp="http://java.sun.com/JSP/Page"
  xmlns:c="http://java.sun.com/jsp/jstl/core"
  xmlns:base="http://calyxo.odysseus.de/jsp/base"
  version="2.0">
<jsp:directive.page contentType="text/html"/>
<base:access var="calyxo"/>
<html>
  <head>
    <title>Login page</title>
  </head>
  <body>
    <h3>Login, please...</h3>
    <base:form action="/login">
      <table>
        <tr>
          <td align="right">User Id</td>
          <td><input type="text" name="user"/></td>
        </tr>
        <tr>
          <td align="right">Password</td>
          <td><input type="password" name="password"/></td>
        </tr>
      </table>
      <input type="submit" value="Submit"/>
    </base:form>
    <c:if test="${!empty calyxo.control.errors}">
      <h3>Action errors</h3>
      <ul>
        <c:forEach var="message" items="${calyxo.control.errors.allMessages}">
          <li>${calyxo.base.i18n.format[message]}</li>
        </c:forEach>
      </ul>
    </c:if>
  </body>
</html>
</jsp:root>
```

The page declares the JSTL core tag library with prefix `c` and the *Calyxo Base* tag library with prefix `base`. The tags from that library are:

- `<base:access var="calyxo"/>` This tag installs a hierarchy of accessors in request scope attribute `calyxo`, which can be used to navigate most of the *Calyxo*-related information. In our document, we use it to access and format our action error messages.
- `<base:form action="/login">...</base:form>` This tag is a wrapper for the HTML form tag. It takes an action path (within the current module) as an attribute. When rendered as HTML, it converts the action path to a context-relative path selecting that action.

Welcome page

Fortunately, the welcome JSP page is much shorter. It accesses the user name from session scope and provides a link to the logout action.

```
<jsp:root
  xmlns:jsp="http://java.sun.com/JSP/Page"
  xmlns:c="http://java.sun.com/jsp/jstl/core"
  xmlns:base="http://calyxo.odysseus.de/jsp/base"
  version="2.0">
<jsp:directive.page contentType="text/html"/>
<base:access var="calyxo"/>
<html>
  <head>
    <title>Welcome page</title>
  </head>
  <body>
    <h3>Welcome, ${user}!</h3>
    Of cause, you can
    <c:url var="href" value="${calyxo.base.module.path['/logout']}" />
    <a href="${href}">logout</a> again.
  </body>
</html>
</jsp:root>
```

Again, this page uses the `<base:access>` tag. This time, it is used to construct a URL to point to the `/logout` action. The expression `${calyxo.base.module.path['/logout']}` evaluates to the context-relative path of that action. For example, if our module uses extension mapping `*.do`, the result is `/logout.do`. If we used a prefix mapping like `/main/*`, the result would have been `/main/login`. Finally, the JSTL core `<c:url>` tag prepends the context name (and may do URL rewriting).

However, as shown in the logout page, there's an even easier way to link to actions using the `<base:a>` tag.

Goodbye page

The goodbye JSP page has nothing new. It just provides a link to login again.

```
<jsp:root
  xmlns:jsp="http://java.sun.com/JSP/Page"
  xmlns:base="http://calyxo.odysseus.de/jsp/base"
  version="2.0">
<jsp:directive.page contentType="text/html"/>
<html>
  <head>
    <title>Goodbye page</title>
  </head>
  <body>
```

```

        <h3>Goodbye!</h3>
        Feel free to <base:a action="/index">login</base:a> again.
    </body>
</html>
</jsp:root>

```

This time - as promised - we used Calyxo's `<base:access>` anchor tag to provide the link to our action.

1.3.2.2. Providing localized messages

Our login action produced error messages. Of course, we have to provide the corresponding resources to be able to show up textual messages to the user.

Let's recover the code parts in our login action:

```

if (!users.containsKey(user)) {
    Message.Arg arg = new Message.ValueArg(user);
    Message message = new Message("messages", "login.user.unknown", arg);
    MessageUtils.saveError(request, "user", message);
    return getActionConfig().getSourceDispatchConfig();
}
if (!users.get(user).equals(request.getParameter("password"))) {
    Message message = new Message("messages", "login.failed");
    MessageUtils.saveError(request, message);
    return getActionConfig().getSourceDispatchConfig();
}

```

In the first code block, we reference a message resource with bundle name `messages` and resource key `login.user.unknown`. Furthermore, this message takes the (unknown) user name as a parameter.

In the second block, we reference a message resource with the same bundle name and resource key `login.failed`. This message takes no parameters.

Since messages use Java's `java.util.ResourceBundle` class to resolve messages, we simply have to place a `messages.properties` file into our classpath, containing messages in our default language. That is, save the following as `/WEB-INF/classes/messages.properties`:

```

login.user.unknown = User id {0} is unknown.
login.failed = Login failed.

```

That's it. Adding messages for other languages easy. For example, if you wanted to add a german message file, save it to `/WEB-INF/classes/messages_de.properties`.

1.3.3. Final tasks

We're now ready to deploy and run our sample login application. There's only very little left to do...

Creating the default index page

When users access the application for the first time, they usually point their browser to the context, not to a particular page or action. In our deployment descriptor, we had the following lines:

```
<!-- Welcome File -->
<welcome-file-list>
  <welcome-file>index.jsp</welcome-file>
</welcome-file-list>
```

We want to create a page that simply passes control over to our /index action, which acts as our application entry point. Here's the JSP page to be saved as /index.jsp:

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0">
  <!-- forward to our start action -->
  <jsp:forward page="/index.do"/>
</jsp:root>
```

Note

Since the welcome page is directly invoked by the servlet container, there's no current module, yet. This is the one and only place you need to use a full path with module extension (or prefix).

Copying classes and libraries

Now, we need to copy our application class tree to /WEB-INF/classes.

Finally, we have to copy the required libraries into /WEB-INF/lib. These are

- CALYXO_HOME/lib/commons-*.jar
- CALYXO_HOME/lib/web/log4j-*.jar
- CALYXO_HOME/lib/web/jstl-1.1/*.jar
- CALYXO_HOME/calyxo-base/calyxo-base-*.jar
- CALYXO_HOME/calyxo-control/calyxo-control-*.jar

We also need a configuration file for log4j. Save the following to /WEB-INF/classes/log4j.properties:

```
### direct log messages to stdout ###
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target=System.out
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE} %5p %-18c{1} - %m%n

### set log levels - for more verbose logging change 'info' to 'debug' ###
log4j.rootLogger=warn, stdout
log4j.logger.de.odysseus.calyxo=info
```

Your web application directory tree should now look like shown below (version numbers may

differ):

```
login-sample/
  WEB-INF/
    classes/
      de/odysseus/calyxo/sample/login/
        LoginAction.class
        LogoutAction.class
      messages.properties
      log4j.properties
    jsp/
      goodbye.jspx
      login.jspx
      welcome.jspx
    lib/
      calyxo-base-0.9.0.jar
      calyxo-control-0.9.0.jar
      commons-beanutils-1.7.0.jar
      commons-digester-1.7.jar
      commons-el-1.0.jar
      commons-logging-1.0.4.jar
      log4j-1.2.8.jar
      jstl.jar
      standard.jar
      calyxo-control-config.xml
      web.xml
  index.jspx
```

1.3.4. Deploying the application

Web application deployment varies depending on the servlet container or application server you use. Please, consult the appropriate documentation.

In case your server requires a web application archive (war) file for deployment, we'll show you how to create it:

1. Change directory into login-sample
2. Execute `jar cvf login-sample.war *`

During deployment you may be asked for a context name. Probably, login-sample is a good choice.

Tomcat 5 users

Tomcat 5 users may simply copy either the login-sample directory or the login-sample.war file to TOMCAT_HOME/webapps.

Does it work?

Open a web browser and point it to `http://<hostname>:<port>/login-sample` and see what happens...

1.3.5. Splitting the Application into Modules

Now, we'll divide our login sample application into two modules. The first module - called *outside* - will provide the part of the application, where the user is *not* logged in. The second module - called *inside* - will provide the part, where the user is logged in. To achieve this, we need to

1. split our controller configuration into the two files `calyxo-control-config-outside.xml` and `calyxo-control-config-inside.xml`.
2. Adjust the deployment descriptor, `web.xml`, to contain the required servlet declarations and mappings for our modules.
3. Optionally, rearrange resource locations to reflect the new module structure. Also, we may want to avoid `118n` resource bundles to be shared by different modules.

Before we show you how to do this, let us consider an important issue on modules.

Switching modules

Though an application should be partitioned into modules with minimizing dependencies in mind, there's obviously a need to change the current module. Basically, *Calyxo* supports two ways to do that:

- In a dispatch element of a module's controller configuration, specify both, the module and action attributes, to dispatch to an action in another module.
- In a JSP page, jump to another module via a HTML link. Again, specify the module and action attributes in *Calyxo's* `<calyxo-base:a>` tag.

This is a nice thing, because your module does not have to know anything about how modules are mapped to url patterns. All you have to know is the target module's name and the action path. *Calyxo* will do the rest for you.

1.3.5.1. Splitting the configuration

Now we know enough to create the controller configurations for our modules. The *outside* module hosts the `login.jsp` and `goodbye.jsp` pages, because they're presented to users, who are not logged in. Here's `calyxo-control-config-outside.xml`:

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

  <actions>

    <!-- The index action just forwards to our login page -->
    <action path="/index">
      <dispatch path="/WEB-INF/jsp/login.jspx"/>
    </action>
```

```

<!-- This action shows the goodbye page -->
<action path="/goodbye">
  <dispatch path="/WEB-INF/jsp/goodbye.jsp"/>
</action>

<!-- Login action -->
<action path="/login" class="de.odysseus.calyxo.sample.login.LoginAction">
  <dispatch name="success" module="inside" action="/index"/>
  <dispatch name="input" path="/WEB-INF/jsp/login.jsp"/>
</action>

</actions>

</calyxo-control-config>

```

As you can see, we switch to module *inside* in the *success* dispatch element of the `/login` action.

`calyxo-control-config-inside.xml` looks like this:

```

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

  <actions>

    <!-- The index action just forwards to our welcome page -->
    <action path="/index">
      <dispatch path="/WEB-INF/jsp/welcome.jsp"/>
    </action>

    <!-- Logout action -->
    <action path="/logout" class="de.odysseus.calyxo.sample.login.LogoutAction">
      <dispatch name="success" module="outside" action="/goodbye"/>
    </action>

  </actions>

</calyxo-control-config>

```

With the *success* dispatch of our `/logout` action, we switch back to module *outside*.

1.3.5.2. Adjusting the deployment descriptor

The deployment descriptor `/WEB-INF/web.xml` needs some straight forward modifications: add a second module servlet and servlet mapping. Here's the new version:

```

<web-app xmlns="http://java.sun.com/xml/ns/j2ee" version="2.4">

  <!-- Module Servlets -->

```

```

<servlet>
  <servlet-name>outside</servlet-name>
  <servlet-class>de.odysseus.calyxo.control.ModuleServlet</servlet-class>
  <init-param>
    <param-name>config</param-name>
    <param-value>/WEB-INF/calyxo-control-config-outside.xml</param-value>
  </init-param>
  <load-on-startup/>
</servlet>
<servlet>
  <servlet-name>inside</servlet-name>
  <servlet-class>de.odysseus.calyxo.control.ModuleServlet</servlet-class>
  <init-param>
    <param-name>config</param-name>
    <param-value>/WEB-INF/calyxo-control-config-inside.xml</param-value>
  </init-param>
  <load-on-startup/>
</servlet>

<!-- Module Mappings -->
<servlet-mapping>
  <servlet-name>outside</servlet-name>
  <url-pattern>*.do</url-pattern>
</servlet-mapping>
<servlet-mapping>
  <servlet-name>inside</servlet-name>
  <url-pattern>/inside/*</url-pattern>
</servlet-mapping>

<!-- Welcome File -->
<welcome-file-list>
  <welcome-file>index.jsp</welcome-file>
</welcome-file-list>
</web-app>

```

We choose a prefix mapping for module *inside*, just for fun.

Note

If we also changed the servlet mapping for the *outside* module, we needed to adjust the `<jsp:forward>` in our welcome file, `/index.jsp`.

Test it, now (you may need to redeploy/restart/reload the application, depending on the application server you use). It should behave exactly as before.

1.3.6. Using Panels

Now let's give our application a homogeneous user interface. We'll define a layout template

shared by all views and use the panels dispatcher to compose our pages dynamically from the templates. This ensures consistency, removes redundancy and greatly improves maintainability. You should have read the [introduction to Calyxo Panels](#) component, before you proceed.

To enable panels for our sample application, we need to

- provide panel definition files for our modules
- create JSP templates, referenced by panel definitions
- adjust our controller configuration to use panels

Yet simple, our layout will use a *cascading stylesheet* (CSS) to define layout details. That is, we will have to provide that css file, too.

We'd like to present the panel definitions first, though the referenced JSP templates don't exist, yet. This way, we know exactly, which templates we need to write.

Warning

This is not a course on web design! Our design reflects the need to keep the example short and simple to remain presentationable.

Before we forget: copy CALYXO_HOME/calyxo-panels/calyxo-panels-*.jar to /WEB-INF/lib now.

1.3.6.1. Panels Configuration

We have to provide panel configuration files containing our panel definitions for each module. Since we want to share a basic layout throughout our application, we'll define a common configuration file, which will be imported by each module's panel configuration file.

Shared layout

We'll use a "classic" layout, which decomposes a page into header, menu, content and footer parts. We'll save this configuration as /WEB-INF/calyxo-panels-shared.xml.

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

    <!-- abstract page layout panel. Concrete subpanels need to
      specify a page title and content. Optionally they may
      provide menu items. -->
    <panel name="/layout.page" template="/WEB-INF/jsp/page.jsp">
      <!-- title (abstract) -->
      <param name="title"/>
      <!-- header -->
      <panel name="header" template="/WEB-INF/jsp/header.jsp">
        <param name="text" value="Login Sample App"/>
      </panel>
    </panel>
  </panels>
</calyxo-panels-config>
```

```

</panel>
<!-- menu -->
<panel name="menu" template="/WEB-INF/jsp/menu.jsp">
  <list name="items">
    <!-- subpanels may define menu items like this:
    <item>
      <param name="title" value="Logout"/>
      <param name="action" value="/logout"/>
    </item>
    -->
  </list>
</panel>
<!-- content (abstract) -->
<panel name="content"/>
<!-- messages -->
<panel name="messages" template="/WEB-INF/jsp/messages.jsp"/>
<!-- footer -->
<panel name="footer" template="/WEB-INF/jsp/footer.jsp"/>
</panel>

</panels>
</calyxo-panels-config>

```

DTD lookup

Many XML editors allow to associate root element names with DTDs. If your editor supports this, you may want to associate calyxo-panels-config with CALYXO_HOME/calyxo-panels/conf/share/calyxo-panels-config.dtd. Alternatively, you should adjust the DTD system path or simply copy the DTD file to your /WEB-INF directory.

This file defines the single toplevel panel /layout.page containing

- A title parameter, which will be used as the page title. The parameter value is left undefined.
- A header panel, which takes a text parameter. The corresponding template is /WEB-INF/jsp/header.jsp.
- A menu panel, which contains a list of items, each having title and action parameters. However, as defined here, the list is empty. The corresponding template is /WEB-INF/jsp/menu.jsp.
- A content panel. This panel is abstract, since it does not specify a template.
- A footer panel, taking no parameters. The corresponding template is /WEB-INF/jsp/footer.jsp.

Derived panels will have to provide a value for the title parameter and a template attribute for the content panel. Optionally, they may override the empty list of items in the menu panel.

Panels for module "outside"

Our outside module has to provide concrete panel definitions for the login - and goodbye

pages. To specify a common header text and menu items used within the module, we define an abstract base panel which will be extended by concrete panels. The base panel extends our shared layout panel we defined above. We'll save this configuration as `/WEB-INF/calyxo-panels-outside.xml`.

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

  <base:import file="calyxo-panels-config-shared.xml"/>

  <panels>

    <!-- base page for pages in this module -->
    <panel name="/base.page" super="/layout.page">
      <panel name="header">
        <param name="text" value="Login Sample App (outside)"/>
      </panel>
      <panel name="menu">
        <list name="items">
          <item>
            <param name="title" value="Login"/>
            <param name="action" value="/index"/>
          </item>
        </list>
      </panel>
    </panel>

    <!-- login page -->
    <panel name="/login.page" super="/base.page">
      <param name="title" value="Login page"/>
      <panel name="content" template="/WEB-INF/jsp/login.jsp"/>
    </panel>

    <!-- goodbye page -->
    <panel name="/goodbye.page" super="/base.page">
      <param name="title" value="Goodbye page"/>
      <panel name="content" template="/WEB-INF/jsp/goodbye.jsp"/>
    </panel>

  </panels>
</calyxo-panels-config>
```

As you can see, we use the `<base:import>` element to include the configuration file containing our shared layout definition.

The `/base.page` panel still leaves the page title and content template open to the `/login.page` and `goodbye.page` panels.

Panels for module "inside"

Due to the nature of our sample application, the inside module has to provide only one concrete panel definition for its welcome page. We should save this configuration as /WEB-INF/calyxo-panels-outside.xml.

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

  <base:import file="calyxo-panels-config-shared.xml"/>

  <panels>

    <!-- base page for pages in this module -->
    <panel name="/base.page" super="/layout.page">
      <panel name="header">
        <param name="text" value="Login Sample App (inside)"/>
      </panel>
      <panel name="menu">
        <list name="items">
          <item>
            <param name="title" value="Logout"/>
            <param name="action" value="/logout"/>
          </item>
        </list>
      </panel>
    </panel>

    <!-- welcome page -->
    <panel name="/welcome.page" super="/base.page">
      <param name="title" value="Welcome page"/>
      <panel name="content" template="/WEB-INF/jsp/welcome.jsp"/>
    </panel>

  </panels>

</calyxo-panels-config>
```

Again, we `<base:import>` our shared layout configuration file and define a `/base.page` panel specifying header text and menu items.

The `/base.page` panel still leaves the page title and content template open to the `/welcome.page` panel.

1.3.6.2. JSP templates

Our panel definitions tell us, what JSP templates we have need to create. A template can

- include a subpanel using the `<panels:panel name="...">` tag,
- access a panel parameter using the a JSTL expression like `${calyxo.panels.param['...']}`,
- iterate over a list using the `<panels:list name="...">` tag.

Page layout template

The `/WEB-INF/jsp/page.jsp` template contains our page layout. Here's the code:

```

<!-- Simple layout: title, header, menu, body, footer -->
<jsp:root version="2.0"
  xmlns:base="http://calyxo.odysseus.de/jsp/base"
  xmlns:panels="http://calyxo.odysseus.de/jsp/panels"
  xmlns:jsp="http://java.sun.com/JSP/Page">
  <jsp:directive.page language="java" contentType="text/html"/>
  <!-- since we define the calyxo access here,
        included templates don't need to redefine it. -->
  <base:access var="calyxo"/>
  <html>
    <head>
      <title>${calyxo.panels.param.title}</title>
      <base href="${calyxo.base.context.home}/index.jsp"/>
      <a href="style.css" type="text/css" rel="stylesheet"/>
    </head>
    <body>
      <table width="100%" cellpadding="0" cellspacing="4">
        <tr>
          <td class="header" colspan="2">
            <panels:panel name="header"/>
          </td>
        </tr>
        <tr>
          <td class="menu">
            <panels:panel name="menu"/>
          </td>
          <td class="content">
            <panels:panel name="content"/>
            <div class="messages">
              <panels:panel name="messages"/>
            </div>
          </td>
        </tr>
        <tr>
          <td class="footer" colspan="2">
            <panels:panel name="footer"/>
          </td>
        </tr>
      </table>
    </body>
  </html>

```

```
</jsp:root>
```

The `${calyxo.base.context.home}` expression evaluates to an absolute URL pointing to our application, e.g. `http://foo.bar.com:8080/login-sample`. The HTML base makes sure your browser interprets relative paths as context-relative, not relative to the current request URL.

The template then uses the stylesheet `style.css`, which we'll create later. This file has to be placed into the application's root directory.

In the simple table-based layout, the template renders the page title and includes its subpanels, as expected.

Menu template

The `/WEB-INF/jsp/menu.jsp` template iterates over the list of menu items and displays them as links:

```
<jsp:root version="2.0"
  xmlns:base="http://calyxo.odysseus.de/jsp/base"
  xmlns:panels="http://calyxo.odysseus.de/jsp/panels"
  xmlns:jsp="http://java.sun.com/JSP/Page">
  <ul>
    <panels:list name="items">
      <li>
        <base:a action="${calyxo.panels.param.action}">
          ${calyxo.panels.param.title}
        </base:a>
      </li>
    </panels:list>
  </ul>
</jsp:root>
```

Messages template

The `/WEB-INF/jsp/messages.jsp` template iterates over the action error messages and displays them:

```
<jsp:root version="2.0"
  xmlns:c="http://java.sun.com/jsp/jstl/core"
  xmlns:jsp="http://java.sun.com/JSP/Page">
  <c:if test="${!empty calyxo.control.errors}">
    <h3>Action errors</h3>
    <ul>
      <c:forEach var="message"
        items="${calyxo.control.errors.allMessages}">
        <li>${calyxo.base.i18n.format[message]}</li>
      </c:forEach>
    </ul>
  </c:if>
```

```
</jsp:root>
```

Header and footer templates

The /WEB-INF/jsp/header.jspx template displays the application title:

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0">
  <h1>${calyxo.panels.param.text}</h1>
</jsp:root>
```

The /WEB-INF/jsp/footer.jspx template just displays static text:

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0">
  <div align="left"><em>Powered by Calyxo!</em></div>
</jsp:root>
```

The content templates

Roughly spoken, our content templates are our old JSP pages, stripped by the HTML decoration, that has gone to the page layout template.

We begin with the two templates from module outside. Here's the /WEB-INF/jsp/login.jspx template:

```
<jsp:root
  xmlns:jsp="http://java.sun.com/JSP/Page"
  xmlns:base="http://calyxo.odysseus.de/jsp/base"
  version="2.0">
  <h3>Login, please...</h3>
  <base:form action="/login">
    <table>
      <tr>
        <td align="right">User Id</td>
        <td><input type="text" name="user"/></td>
      </tr>
      <tr>
        <td align="right">Password</td>
        <td><input type="password" name="password"/></td>
      </tr>
    </table>
    <input type="submit" value="Submit"/>
  </base:form>
</jsp:root>
```

The /WEB-INF/jsp/goodbye.jspx template is trivial:

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0">
  <h3>Goodbye!</h3>
</jsp:root>
```

The same applies to the /WEB-INF/jsp/welcome.jspx template used by module inside:

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0">
  <h3>Welcome, ${user}!</h3>
</jsp:root>
```

1.3.6.3. Adjusting the controller configuration

Now, that we defined our panels and implemented the corresponding templates, we have to tell the controller to use them. This is done by loading the *Calyxo Panels* plugin.

In `/WEB-INF/calyxo-control-config-outside.xml` and `/WEB-INF/calyxo-control-config-inside.xml` we have to change occurrences of `.jspx` to `.page` and add a `<plugin>` element to load the panels plugin. Here's the modified `/WEB-INF/calyxo-control-config-outside.xml`:

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

  <plugins>

    <!-- Install panels dispatcher as global dispatcher -->
    <plugin class="de.odysseus.calyxo.panels.control.PanelsPlugin">
      <param name="config" value="/WEB-INF/calyxo-panels-config-outside.xml"/>
      <param name="global" value="true"/>
    </plugin>

  </plugins>

  <actions>

    <!-- The index action just forwards to our login page -->
    <action path="/index">
      <dispatch path="/login.page"/>
    </action>

    <!-- This action shows the goodbye page -->
    <action path="/goodbye" target="default">
      <dispatch path="/goodbye.page"/>
    </action>

    <!-- Login action -->
    <action path="/login" class="de.odysseus.calyxo.sample.login.LoginAction">
      <dispatch name="success" module="inside" action="/index"/>
      <dispatch name="input" path="/login.page"/>
    </action>

  </actions>

</calyxo-control-config>
```

The global parameter tells the plugin to install itself as the default dispatcher for that module. However, you can define a different dispatcher for an action or even for a single dispatch element.

Make analog changes to `/WEB-INF/calyxo-panels-config-inside.xml` and - you're done!
Well, almost...

1.3.6.4. Adding style

In fact, the application should run. Try it (you may need to redeploy/restart/reload the application, depending on the application server you use).

Yes, but it - ooh - looks still bad!

Our `/WEB-INF/jsp/page.jsp` template misses the CSS stylesheet, which we'll add now to complete our *Calyxo Panels* example. Save the following to `/style.css`:

```
body {
    background-color: white;
    margin: 0px;
    padding: 10px;
    font-family: Verdana, Helvetica, sans-serif;
}

.menu {
    padding: 10px;
    border-right: thin dotted gray;
    width: 15%;
    height: 450px;
    vertical-align: top;
    font-size: 10pt;
}
.menu ul {
    padding: 0px;
    margin: 0px;
}
.menu li {
    margin-bottom: 10px;
    list-style: none;
    font-weight : bold;
}
.menu a:link { color: gray; text-decoration : none; }
.menu a:visited { color: gray; text-decoration : none; }
.menu a:hover { color: black; text-decoration : none; }

.header {
    border-bottom: thin dotted gray;
    height: 80px;
}
```

```
.content {
  padding: 10px;
  padding-left: 20px;
  vertical-align: top;
  font-size: 10pt;
}
.content td, h3 {
  font-size: 10pt;
}

.footer {
  border-top: thin dotted gray;
  font-size: 8pt;
}
```

Again, test your application. It should look much better now!

1.3.7. Validating Forms

We'll take the login form to demonstrate the use of *Calyxo's* form validation capabilities. The entered login data will be validated according to the following conditions:

- A user name is valid if it matches the regular expression `^[a-z][a-zA-Z0-9]{2,7}$`. That is, it must start with a letter, followed by two to seven alpha-numerical characters.
- The password must not be empty.
- The user id and password must not be equal.

Note

In fact, the latter condition may seem somewhat strange. Don't mind! We just want to demonstrate the assertion of a condition depending on multiple fields.

All we have to do, is to describe the above conditions in a forms configuration file, load the forms plugin and change the `/WEB-INF/jsp/login.jsp` template to use the `<forms:form>`, `<forms:text>` and `<forms:password>` tags.

Note

The capabilities of the *Calyxo Forms* component go far beyond of what we use in this example. We do not use converters and checkers, here; neither we use form data beans; neither we define custom validators; ...

Before we forget: copy `CALYXO_HOME/calyxo-forms/calyxo-forms-*.jar` to `/WEB-INF/lib` now.

1.3.7.1. Forms Configure

We'll create a forms configuration file for our outside module. The file will contain a form definition for our login form. Save the following to

/WEB-INF/calyxo-forms-config-outside.xml:

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

  <forms>

    <form name="login">
      <field property="user">
        <match name="regex">
          <property name="pattern" value="^[a-z][a-zA-Z0-9]{2,7}$"/>
          <message bundle="messages" key="login.user.invalid"/>
        </match>
      </field>
      <field property="password">
        <match name="notEmpty">
          <message>
            <arg name="field" value="password"/>
          </message>
        </match>
      </field>
      <assert test="input.user != input.password">
        <message bundle="messages" key="login.userAndPasswordEqual"/>
      </assert>
    </form>

  </forms>

</calyxo-forms-config>
```

The configuration defines a form named login with two input fields:

- the user field contains a match element, which uses the predefined regex matcher. We specify the pattern to be used with the property element. We also have a message element, with both, the bundle and key attributes set. This overrides the generic message from the regex matcher.
- the password field uses the predefined notEmpty matcher. In the contained message element, it passes a value for the field argument to the message defined by the notEmpty matcher.
- The assert expression takes an EL expression in its test attribute. It fails, if the expression evaluates to false, that is, if the user id and password are equal.

1.3.7.2. Adjusting the controller configuration

Now, that we defined our forms, we need to adjust our controller configuration to perform validations.

In /WEB-INF/calyxo-control-config-outside.xml we have to add a <plugin> element to load the forms plugin. Add the following to the <plugins> element:

```

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

```

The forms plugin provides a *filter*, which may be used by actions to trigger form validation. You tell the filter about what form definition to apply and what dispatch configuration to use when validation fails. Change the /login action to the following:

```

<!-- Login action -->
<action path="/login" class="de.odysseus.calyxo.sample.login.LoginAction">
  <filter name="forms">
    <param name="form" value="login"/>
    <param name="dispatch" value="input"/>
  </filter>
  <dispatch name="success" module="inside" action="/index"/>
  <dispatch name="input" path="/login.page"/>
</action>

```

1.3.7.3. Modifying the view

So far, validation is enabled. In fact, if you'd restart your application, validations would be performed. Try it, if you like.

However, when validation fails, we want to give the user some visual feedback. We'd like form controls corresponding to invalid fields to be marked red. If an assert condition fails, we'd want the controls involved to be marked, too. Even more important, we want to redisplay the inputs, the user made, when validation fails.

To get all this, you only have to use the custom tags, that come with the *Calyxo Forms* component. So, change the content of /WEB-INF/jsp/login.jsp as follows:

```

<jsp:root
  xmlns:jsp="http://java.sun.com/JSP/Page"
  xmlns:base="http://calyx.odyseus.de/jsp/base"
  xmlns:forms="http://calyx.odyseus.de/jsp/forms"
  version="2.0">
  <h3>Login, please...</h3>
  <forms:form action="/login">
    <table>
      <tr>
        <td align="right">User Id</td>
        <td><forms:text name="user"/></td>
      </tr>
      <tr>
        <td align="right">Password</td>
        <td><forms:password name="password"/></td>
      </tr>
    </table>

```

```

        <input type="submit" value="Submit"/>
    </forms:form>
</jsp:root>

```

The last thing to do is adding error messages to our `/WEB-INF/classes/messages.properties` file:

```

login.user.unknown = User id {0} is unknown.
login.failed = Login failed.

# validation messages
login.user.invalid = Invalid user id.
login.userAndPasswordEqual = User id and password must not be equal.

```

You're done. Now, restart the application and see if form validation works.

1.3.8. Using Struts as Controller

In our sample application, we used the *Calyxo Control* component. However, you may also use Struts as controller. We won't provide the exact code for our sample application, but here's a sketch of changes to be made:

- Provide Struts configuration files analogous to the *Calyxo* variants. Additionally, load the *Calyxo Base*, *Calyxo Panels* and *Calyxo Forms* plugins.
- Add a form bean definition for the login form.
- Implement Struts actions instead of *Calyxo* actions.
- In your `web.xml`, define a single Struts action servlet with module configurations instead of *Calyxo* module servlets.
- Replace the `messages.jsp` template with a variant that accesses Struts action errors.

Please refer to the *Calyxo Struts* component for further details.

1.4. HOWTO's

1.4.1. Calyxo Eclipse Plugins

The following *Calyxo* Eclipse plugins are available:

- The `de.odysseus.calyxo.eclipse.help` plugin adds the complete *Calyxo* documentation to the Eclipse Help system. After installation, the *Calyxo* manual can be browsed and searched via *Help # Help Contents*.
- The `de.odysseus.calyxo.eclipse.wst.xml` plugin provides XML Catalog extensions for the *Calyxo* XML Schemas and tag library descriptors. After installation, the XML and JSP editors support validation and content assist when editing *Calyxo* configuration files or JSP files which use the *Calyxo* tag libraries. Requires the *Eclipse Web Tools Platform*.

The following instructions describe how to install the plugins via Eclipse update manager:

1. Go to *Help # Software Updates # Find and Install...*

2. Select *Search for new features to install*. Click *Next*.
3. Click *New Remote Site*. Enter *Odysseus Update Site* as name and <http://odysseus.de/calyxo/eclipse/updates> as URL. Click *OK*.
4. You should now see a new entry *Odysseus Update Site* with a mark next to it. Check it and press *Finish*.
5. Expand *Odysseus Update Site # Calyxo MVC Web Application Framework*. Check *Calyxo IDE 0.9.0* and click *Next*.
6. Read the license agreement. Select *I accept...* and click *Next*.
7. Click *Finish* to start the installation. Click *Install* on the warning dialogs during feature verification.

After restarting Eclipse you can verify that the new feature is available by going to *Help # About Eclipse SDK* and clicking on the *Calyxo* logo.

1.4.2. Deployment

There's nothing special about the deployment of a *Calyxo* web application. However, you'll need to copy the required libraries from the *Calyxo* binary distribution to your web application's `/WEB-INF/lib` directory.

- `CALYXO_HOME/lib/commons-*.jar`
required Apache Jakarta Commons: digester, beanutils, el, logging
- `CALYXO_HOME/lib/web/log4j-*.jar`
if you're using the Apache Log4J logging engine
- `CALYXO_HOME/lib/web/jstl-1.1/*.jar`
if your container doesn't provide JSTL libraries (e.g. Tomcat 5)
- `CALYXO_HOME/calyxo-struts/lib/struts-*.jar`
if you're using Struts (requires 1.2.4 or better)
- `CALYXO_HOME/calyxo-base/calyxo-base-*.jar`
the *Calyxo Base* classes are always required
- `CALYXO_HOME/calyxo-control/calyxo-control-*.jar`
if you're using *Calyxo Control*
- `CALYXO_HOME/calyxo-panels/calyxo-panels-*.jar`
if you're using *Calyxo Panels*
- `CALYXO_HOME/calyxo-forms/calyxo-forms-*.jar`
if you're using *Calyxo Forms*
- `CALYXO_HOME/calyxo-struts/calyxo-struts-*.jar`
if you're using *Calyxo Struts*

1.4.3. Log4J Configuration

If you didn't use log4j before, you may want to get started by saving the following to `/WEB-INF/classes/log4j.properties`:

```
### direct log messages to stdout ###
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target=System.out
```

```
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE} %5p %-18c{1} - %m%n

### set log levels - for more verbose logging change 'info' to 'debug' ###
log4j.rootLogger=warn, stdout
log4j.logger.de.odysseus.calyxo=info
```

2. Calyxo Base

Calyxo has been designed to ease the development of JSP Model 2 applications. In a Model 2 application servlets take over the controller part and JSPs are used as view technology.

In particular, *Calyxo* has been designed to be used with controllers, that implement the *Command and Controller* strategy. This strategy builds on the *Front Controller* and *Application Controller* design patterns. The *Front Controller*, implemented as a servlet, acts as a centralized access point for incoming requests. It delegates to an *Application Controller (module)*, which is responsible for identifying and invoking *Command Objects (actions)* and for identifying and dispatching to views.

However, the *Calyxo Base* component does not implement a controller. It just *assumes* a controller following the above strategy by introducing *modules* and *actions* in an abstract manner.

The *Calyxo Base* component factors out some of the fundamental patterns and services supported by the *Calyxo* framework, which are used by applications as well as throughout all the other components.

Application Modules

Calyxo applications may be separated into (more or less) independent *modules*.

It is central to modules as used by *Calyxo*, that they are basically containers for so called *actions*. Struts users will be familiar to the concept of actions. They are the entry points of a module: a module handles a request by selecting an action by its path, executing it and - depending on the action's result - dispatching to another action or resource.

Modularization is a proven strategy to break down complexity and to ease in team development. In pattern terminology, a module realizes the *Application Controller* pattern.

Powerful Configuration Capabilities

Calyxo components are configured by XML files, which all share common abilities, such as importing other configuration files, defining variables, using JSP EL expressions, storing objects, etc. This leads to high expressive configuration formats and unifies basic features across the platform, thus simplifying their use.

New Accessors Presentation Model

Traditionally, *custom tags* are used to access logic and data from within JSP pages. With the integration of the expression language (EL) into the JSP 2.0 standard, many custom tags have become obsolete. Consequent usage of EL expressions in combination with the Java Standard Tag Library (JSTL) may now lead to very nice JSP code. *Calyxo* supports this approach by

providing so-called *accessors*, that present themselves as a hierarchy of beans and maps, ready to be used in EL expressions.

Calyxo's accessors realize the *Presentation Model* and *View Helper* design patterns.

Internationalization Support

In today's application development, i18n has become an important issue. *Calyxo* supports i18n from the ground up. *Calyxo Base* provides the basic pieces as a fundament to application localization. Other *Calyxo* components provide higher level features such as locale-dependent selection of views and forms.

2.1. Concepts

Calyxo itself follows a component based approach. This means, parts of *Calyxo* may be used alone. For example, the *Calyxo Panels* and *Calyxo Forms* components may be used with Struts. However, all *Calyxo* components share some common concepts and features, implemented in the *Calyxo Base* component.

The *Calyxo Base* component collects some of the basic, reusable classes used throughout all the other subprojects. It introduces basic concepts like modules, i18n, accessors and so on...

Modules

Calyxo supports the concept of *modules*. An application is composed of modules, which, in principle, are independent from each other but may choose to share code and data in several ways.

The *Calyxo Base* component introduces modules on a high level by defining an abstract *module context* which grants access to the module's name, scope and initialization parameters, as well as transforming an action path to a context-relative path. Just enough, to get other frameworks, like Struts, on board...

i18n

At the lower end, localizing views requires looking up resources and formatting message in a locale dependent manner. A resource is identified by a bundle name and a resource key. Other *Calyxo* components provide higher level features such as localized views and forms.

Configuration

Calyxo components are configured by XML files, which all share common abilities, such as importing other configuration files, defining variables, using JSP EL expressions, storing objects, etc. This leads to high expressive configuration formats and unifies basic features across the platform, thus simplifying their use. The *Calyxo Base* component provides an API used by other components to participate in that.

Accessors

Calyxo propagates extensive use of JSP EL and JSTL in views instead of implementing custom tags for every beep and whistle. Calyxo Base supports this approach by introducing *accessors*, that present themselves as a hierarchy of beans and maps, ready to be used in EL expressions.

Beside implementing various concrete accessors, the Calyxo Base component provides an API, which may be used by applications to contribute their own accessors.

2.1.1. Modules

Calyxo applications are composed of *modules*. Modules are independent units, you may think of them as subapplications. A module is a container for *actions*.

When handling a request the controller will have to perform three major steps:

1. select a module
2. select an action inside that module and invoke it
3. dispatch the request to another action or resource

The way modules are selected, may vary. For example, the *Calyxo Control* component uses one servlet per module. That way, selecting the appropriate module is left to the servlet container. This approach guarantees a high degree of independence between modules. On the other hand, Struts uses a single servlet per application, which invokes the request processor for the appropriate module.

Current Module

Once a module has been selected, we call it the *current module*.

The current module then selects an *action* and executes it. We say, "*the action is invoked by the request*". An action's execution results in information on how to *dispatch* the request. The module then dispatches to another action (optionally within another module) or to an application resource (for example, a JSP page).

It is important to understand, that the current module exists only during the lifetime of the request, that selected the module. If you, for example, point your browser directly to a JSP page, there's no current module! If, on the other hand, an action dispatches to that page, the action's module will still be the current module within the page.

Module Context

A module is represented by its *context*. The module context provides access to module properties and services, like

- the module name
- the surrounding servlet context

- module initialization parameters
- module scope attributes
- the context relative path for a given action (module relative path).
- a class loader used to load user-defined classes

The interface `de.odysseus.calyxo.base.ModuleContext` defines the corresponding Java type.

As you can see, a module context provides its own attribute scope, just like the servlet context does. Only, it is private to that module. *Calyxo* stores all its configuration information inside module scopes to prevent name clashes between modules.

In a real application, an incoming request has to be mapped to its corresponding module. That is, a context relative path gets decomposed into a module part (identifying the module) and an action path (relative to the module), which is the controller's job. As mentioned above, a module context provides the inverse mapping, answering the question: how do I access a particular action within the module from outside?

Module Support

The `de.odysseus.calyxo.base.ModuleSupport` class knows about the current and all the other modules in the application. There's one instance of this class per application. The various static `getInstance(...)` methods are used to get a reference to it. Once you got that instance, you can use the various `getModuleContext(...)` methods to retrieve the current module context (or another module context by name).

Module Accessors

The `base.module.*` accessors provided by *Calyxo Base* enable for easy access to modules from within JSP pages, for example

```
${calyxo.base.module.name}
${calyxo.base.module.attribute['foo']}
${calyxo.base.module.path['/list']}
```

The last example may be used to build URLs pointing to an action. However, *Calyxo* also provides a `<base:a>` tag, that clones HTML's `<a>` tag, but replaces the href attribute with module and action attributes.

2.1.2. Accessors

Traditionally, *custom tags* are used to access logic and data from within JSP pages. With the integration of the expression language (EL) into the JSP 2.0 standard, many custom tags have become obsolete. Consequent usage of EL expressions in combination with the Java Standard Tag Library (JSTL) may now lead to very nice JSP code. *Calyxo* supports this approach by providing so-called *accessors*, that present themselves as a hierarchy of beans and maps, ready to be used in EL expressions.

Beside implementing various concrete accessors, the *Calyxo Base* component provides the

[Access API](#), which may be used by application programmers to contribute their own accessors.

Please refer to the [Accessors reference](#) to explore the accessors provided by *Calyxo Base*.

Using Accessors

The tree of accessors is instantiated and installed into request scope using the `<base:access>` tag, like in

```
<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>
```

By convention, we use `calyxo` to denote the root.

Now, that the accessors have been installed into request scope at attribute `calyxo`, JSP EL expression of the form `calyxo.*` are used to select and invoke accessors.

As an example, to take property `foo` from the bean at attribute `mybean` in module scope, you would use an expressions like `${calyxo.base.module.attribute['mybean'].foo}`.

2.1.3. Internationalization

Calyxo supports *i18n* from the ground up. At the very basic, *Calyxo* provides ways to resolve resources in a locale dependent manner. A resource is identified by a bundle name and a resource key.

A resource may be a template, that expects arguments to be expanded to a *message*.

I18n Support

The `de.odysseus.calyxo.base.I18nSupport` class contains the methods to lookup resources and to format messages. It is also responsible to serve the *desired locale*. There's one instance per module. You can get the `I18nSupport` instance for the current module by invoking one of the various static `getInstance()` methods.

The `getLocale(HttpServletRequest request)` method queries for the desired locale. Throughout your application, you should always use this method to determine the desired locale. Applications may call `setLocale(HttpServletRequest request, Locale locale)` to save the specified locale as the desired locale, usually into session scope.

The `getResource(Locale locale, String bundle, String key)` method is used to lookup a simple localized resource.

The `getMessage(Locale locale, String bundle, String key, Object[] args)` message is used to lookup a message resource (with any number of message arguments).

Bundle names may be mapped to another name with the `setBundleAlias(String alias, String bundle)` method. Subsequent calls to `getResource(...)` and `getMessage(...)` with bundle alias will actually result in lookups for bundle bundle. This is useful to keep your logical bundle names (aliases) independent of implementation issues (like Java resource bundle names and locations). Since this is a common module configuration issue, we give the following configuration snippet as an equivalent to `I18nSupport.getInstance(moduleContext).setBundleAlias(alias, bundle)`:

```
<use>
  <member class="de.odysseus.calyxo.base.I18nSupport">
    <method name="getInstance">
      <arg value="${moduleContext}"/>
    </method>
  </member>
  <method name="setBundleAlias">
    <arg value="alias"/>
    <arg value="bundle"/>
  </method>
</use>
```

Default I18n Support

The default implementation of `I18nSupport` used by a module depends on the module's environment:

- With *Calyxo Control*, the default mechanism for resources is to use Java's `ResourceBundle` class. Messages are expanded using Java's `MessageFormat` class.

Note

However, your application may provide its own or customized mechanism. Simply subclass `I18nSupport` and implement the abstract `lookupResource(...)` and `lookupMessage(...)` methods. Refer to the section on [Module Initialization](#) on how to make the module use your customized `i18n` implementation.

- The *Calyxo Base* Struts plugin provides a `I18nSupport` implementation, which wraps and delegates to Struts' `MessageResources` mechanism. Here, customization is done on the Struts side.

I18n Accessors

The `base.i18n.*` accessors provided by *Calyxo Base* make localizing content very easy, for example

```
${calyxo.base.i18n.locale}
${calyxo.base.i18n.bundle['strings'].resource['user_id']}
${calyxo.base.i18n.bundle['strings'].message['required']['user_id']}
```

As you can see, there's no need for custom tag libraries to localize your content.

2.1.4. Configuration

Generally, *all Calyxo configuration files* are in XML format.

Each module takes its own set of configuration files. Within a module, each *Calyxo* component is configured through individual configuration files. However, the *Calyxo Base* component is somewhat special. Most of its configuration elements may be reused in configurations of other *Calyxo* components.

Namespaces

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/base`.

Because other components may share *Calyxo Base* configuration elements, we have to consider two scenarios:

1. In a standalone *Calyxo Base* configuration file, we use this namespace as the default namespace.

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

2. In another component, say *Foo*, this namespace should be bound to prefix `base` in the root element of the configuration file, as in

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

DTD Limitation

When using DTD validation, the prefix **must** be `base`. Otherwise, an arbitrary prefix is allowed. However, we recommend to use prefix `base` to keep compatibility.

Throughout this documentation, we'll use the configuration elements without prefix, as they were used in a standalone *Calyxo Base* configuration file. Keep in mind that the `base` prefix has to be added, when these elements are reused in configuration files for other *Calyxo* components.

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 Base* document type:

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

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

Document structure

In a standalone *Calyxo Base* configuration file, the root element is `<calyxo-base-config>`. As common to all of *Calyxo*'s configuration files, the root element specifies the `xmlns` and `version` attributes. It may contain `<import>` elements, followed by `<functions>`, `<set>` and `<use>` elements.

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

  <!-- import elements can go here -->

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

</calyxo-base-config>
```

Importing another configuration file

A configuration file may import another file containing some other parts of the component's configuration. The imported configuration is *merged* into the importing configuration.

```
<import file="../calyxo-control-config-shared.xml"/>
```

The imported file must be either

- of the same type as the importing file or
- a *Calyxo Base* configuration file.

This feature also gracefully supports sharing of common configuration parts between modules.

Dynamic Attributes

Where it makes sense, attribute values of configuration elements may contain EL expressions.

We call these attributes *dynamic*. Expressions may refer to constant expressions like `true`, `false`, `1.23+4.56` and `null`.

Additionally, dynamic attributes are evaluated in a context, that makes available several implicit objects. These implicit objects are always available under the following names:

1. `moduleContext` – resolves to the module context instance
2. `moduleScope` – resolves to a map serving module scope attributes
3. `applicationScope` – resolves to a map serving application scope attributes

Other identifiers are searched as local variable (see below), then as module scope attribute, then as application scope attribute.

Finally, *functions* may be declared and used in dynamic attribute expressions. Functions are static methods of a specified class. The element

```
<functions prefix="foo" class="FooFunctions"/>
```

registers all public, static methods of class `FooFunctions` under prefix `foo`.

Now, expression `${foo:bar(module)}` will evaluate to the returned value of the static method `FooFunctions.bar(...)`, passing in the module context as an argument.

Defining Variables

Variables are defined using the `<set>` element and may be subsequently used in dynamic attribute expressions. The required `var` attribute is used to specify the variable name.

A variable may have one of the following visibility scopes

1. *local* – A local variable is visible inside following siblings of the `<set>` element, which defined the variable. Stated another way, a local variable is visible below its declaration down to the end tag of the `<set>`'s parent element.
2. *module* – A variable with module scope is saved as a module context attribute.
3. *application* – A variable with application scope is saved as a servlet context attribute.

A variable's scope may be specified using the optional `scope` attribute. Valid values are `local`, `module` and `application`. The default scope is `local`.

The dynamic value attribute may be used to specify the variable's value. Here's an example for a (local) variable definition:

```
<set var="content" value="/WEB-INF/${moduleContext.name}"/>
```

Assuming module name `"foo"`, the dynamic attribute expression `${content}/bar.jsp` will evaluate to `/WEB-INF/foo/bar.jsp`.

However, specifying the `value` attribute is the most simple way to define a variable value. Alternatively,

- a nested `<object>` element may be used to create and initialize a new object:

```
<set var="list">
```

```

<object class="java.util.ArrayList">
  <method name="add">
    <arg value="foo"/>
  </method>
</object>
</set>

```

creates a new `java.util.ArrayList` and initializes it by calling its `add()` method with "foo" as argument.

- a nested `<member>` element may be used to access an object's method or field:

```

<set var="size">
  <member value="${list}">
    <method name="size"/>
  </member>
</set>

```

answers the result of invoking method `size()` on the object given by expression `${list}`.

Please refer to the [set, use & Co](#) section for further details.

Using Objects

The `<use>` element can be used to invoke methods and set properties of an object referenced by an expression given in the value attribute:

```

<use value="${list}">
  <method name="add">
    <arg value="bar"/>
  </method>
</use>

```

will invoke method `add` on the object given by expression `${list}`, passing "bar" as an argument.

Please refer to the [set, use & Co](#) section for further details.

2.1.4.1. `<set>`, `<use>` & Co

There are several elements, which are used to set variables, create and manipulate objects:

- The `<set>` element defines a variable
- The `<use>` element sets properties and invokes methods on an object
- The `<object>` element creates and uses a new object
- The `<constructor>` element calls a constructor
- The `<member>` element evaluates to a field value or the result of a method invocation
- The `<method>` element invokes a method
- The `<arg>` element specifies a method or constructor argument
- The `<field>` element accesses a field member
- The `<property>` element sets a Java Bean property or puts a key/value pair into a map

The `<set>` and `<use>` elements represent "statements" and do not appear inside any of the above elements.

Element Summary

The `<set>` element uses the `var` and `scope` attributes to specify the variable name and scope. The `scope` attribute is optional. Valid values are `local`, `module` and `application`. The default scope is `local`.

The `<property>`, `<field>` and `<method>` elements require the `name` attribute to specify the property, field or method name.

The `<object>` element requires the `class` attribute to specify the instantiation class.

The `<member>` element requires at least one of the `value` and `class` attributes. The dynamic `value` attribute evaluates to the object whose method or field is to be accessed. The `class` attribute specifies the class used to search for the member to be accessed. If both attributes are specified, the given `value` must be an instance of the given class. If the `class` attribute is omitted, the search class defaults to the given `value`'s class. If the `value` attribute is omitted, only static members of the given class can be accessed.

The `<set>`, `<property>` and `<arg>` elements take their value from the dynamic `value` attribute or one of:

- a nested `<object>` element
- a nested `<member>` element

The `<object>` element may contain a `<constructor>` as its first child.

The `<use>` element specifies its value to use either by a dynamic `value` attribute or a nested `<member>` element as its first child.

The `<use>` and `<object>` elements contain a (mixed) sequence of:

- nested `<property>` elements
- nested `<method>` elements

The `<member>` element contains one of:

- a nested `<field>` element
- a nested `<method>` element

The `<method>` and `<constructor>` elements contain any number of nested `<arg>` elements

Examples

1. Set a variable using the `value` attribute:

```
<set var="content" value="/WEB-INF/${moduleContext.name}"/>
```

2. Set a variable to a new object, then use it:

```
<set var="jeff">  
  <object class="org.foo.bar.Person"/>
```

```

</set>

<use value="{jeff}">
  <property name="name" value="Jefferson"/>
  <method name="addNickname">
    <arg value="Jeff"/>
  </method>
</use>

```

3. Semantically the same, but use the new instance inside <object>:

```

<set var="jeff">
  <object class="org.foo.bar.Person">
    <property name="name" value="Jefferson"/>
    <method name="addNickname">
      <arg value="Jeff"/>
    </method>
  </object>
</set>

```

4. Same as above, but use non-default constructor:

```

<set var="jeff">
  <object class="org.foo.bar.Person">
    <constructor>
      <arg value="Jefferson"/>
    </constructor>
    <method name="addNickname">
      <arg value="Jeff"/>
    </method>
  </object>
</set>

```

5. Set a variable to a new map and add some associations:

```

<set var="map">
  <object class="java.util.HashMap">
    <property name="foo" value="bar"/>
    <property name="foobar">
      <object class="org.foo.bar.Foobar"/>
    </property>
  </object>
</set>

```

6. Set a variable to a method result:

```

<set var="style">
  <member value="{moduleContext}">
    <method name="getInitParameter">
      <arg value="style"/>
    </method>
  </object>
</set>

```

7. Set a variable to a static method result:

```
<set var="now">
  <member class="java.lang.System">
    <method name="currentTimeMillis"/>
  </member>
</set>
```

8. Set a variable to a static field value:

```
<set var="english">
  <member class="java.util.Locale">
    <field name="ENGLISH"/>
  </object>
</set>
```

9. Log a message using Commons Logging:

```
<use>
  <member class="org.apache.commons.logging.LogFactory">
    <method name="getLog">
      <arg value="de.odysseus.calyxo.base.conf.Test"/>
    </method>
  </member>
  <method name="info">
    <arg value="Hello, world!"/>
  </method>
</use>
```

10. Set the server's default locale to java.util.Locale.ENGLISH:

```
<use>
  <member class="java.util.Locale">
    <method name="getDefault"/>
  </member>
  <method name="setDefault">
    <arg>
      <member class="java.util.Locale">
        <field name="ENGLISH"/>
      </member>
    </arg>
  </method>
</use>
```

11. Create and use a number format:

```
<!-- create a number format for german locale -->
<set var="format">
  <member class="java.text.NumberFormat">
    <method name="getInstance">
      <arg>
        <member class="java.util.Locale">
          <field name="GERMAN"/>
        </member>
      </arg>
    </method>
  </member>
</set>
```

```

    </method>
  </member>
</set>

<!-- set maximum fraction numbers -->
<use value="{format}">
  <property name="maximumFractionDigits" value="2"/>
</use>

<!-- format a number -->
<set var="result">
  <member value="{format}">
    <method name="format">
      <arg value="{123.456}"/>
    </method>
  </member>
</set>

```

2.2. Reference

2.2.1. Configuration

The elements described in the following sections are defined within namespace

<http://calyx0.odysseus.de/xml/ns/base>

Standalone *Calyxo Base* configuration files can be included by other configuration files of any *Calyxo* component. If the XML parser doesn't support XML Schema, DTD validation has to be used by declaring the *Calyxo Base* document type as in:

```

<!DOCTYPE calyx0-base-config
  PUBLIC "-//Odysseus Software GmbH//DTD Calyx0 Base 0.9//EN"
  "calyx0-base-config.dtd">

```

To reuse *Calyxo Base* configuration elements in another component, say *foo*, this namespace should be bound to prefix *base* in the root element of the configuration file, as in

```

<calyx0-foo-config version="0.9"
  xmlns="http://calyx0.odysseus.de/xml/ns/foo"
  xmlns:base="http://calyx0.odysseus.de/xml/ns/base">
  ...
</calyx0-foo-config>

```

DTD Limitation

When using DTD validation, the chosen prefix **must** be *base*. Otherwise, an arbitrary prefix is allowed. However, we recommend to use prefix *base* to keep compatibility.

Dynamic attributes (attributes, whose value may contain EL expressions) are evaluated in a

context, that provides several implicit objects. These implicit objects are always available under the following names:

1. `moduleContext` resolves to the module context instance
2. `moduleScope` resolves to a map serving module scope attributes
3. `applicationScope` resolves to a map serving application scope attributes

Other identifiers are resolved to variables and attributes. That is, they are searched as local variable, then as module scope attribute, then as application scope attribute.

Throughout this reference, required attributes appear **strong**. Dynamic attributes appear *emphasized*.

Elements

Name	Description
<code>calyxo-base-config</code>	Root element of a standalone <i>Calyxo Base</i> configuration file.
<code>import</code>	Import another configuration file.
<code>functions</code>	Register a class defining static methods to be used as EL functions.
<code>set</code>	Define and store a variable.
<code>use</code>	Use an object to invoke methods and set properties.
<code>property</code>	Set a Java Bean property (or put an association into a <code>java.util.Map</code>).
<code>object</code>	Create and initialize a new object.
<code>member</code>	Access a method or field member.
<code>field</code>	Get a field value.
<code>method</code>	Invoke a method.
<code>arg</code>	Specify a method argument.

2.2.1.1. The `<calyxo-base-config>` Element

Purpose

The `<calyxo-base-config>` element is the root element of a *Calyxo Base* 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 <code>http://calyx0.odysseus.de/xml/ns/base</code> .
version	NMTOKEN	Required - DTD/Schema version number. Must be 0.9.

Body

The body of the `<calyx0-base-config>` element is defined by the following sequence:

```
(import*, (functions | set | use)*)
```

Related elements

`<import>`, `<functions>`, `<set>`, `<use>`

2.2.1.2. The `<import>` Element**Purpose**

The `<import>` element is used to import another configuration file. The imported configuration is "merged" into the importing configuration.

The imported configuration file must be either

- of the same type as the importing file or
- a *Calyxo Base* configuration file.

Care should be taken to avoid element- and attribute collisions in the importing and imported configurations.

Attributes

Name	Type	Description
file	CDATA	Required - Path to the configuration file to be imported. If the path starts with a slash (/), it is interpreted as a context-relative path. Otherwise, it is interpreted as an absolute URL.

Body

The `<import>` element has no body.

2.2.1.3. The `<functions>` Element**Purpose**

The <functions> element is used to register the public, static methods of a class for use in EL expressions of dynamic attributes.

Attributes

Name	Type	Description
class	CDATA	Required - The fully qualified class name.
prefix	NMTOKEN	Required - The prefix used in expressions to refer to methods of the class given by the class attribute.

Body

The <functions> element has no body.

2.2.1.4. The <set> Element

Purpose

The <set> element is used to define a *variable*. Variables may be subsequently used in dynamic attribute expressions. The required var attribute is used to specify the variable name.

A variable's scope may be specified using the optional scope attribute. Valid values are

1. local – A local variable is visible below its declaration, inside the element tree rooted at the <set>'s parent element.
2. module – A variable with module scope is saved as a module context attribute.
3. application – A variable with application scope is saved as a servlet context attribute.

A variable's scope may be specified using the optional scope attribute. Valid values are local, module and application. The default scope is local. When evaluating dynamic attribute expressions, identifiers are resolved to local, module and application scope variables (in that order).

The variable value may be specified either by the value attribute, a nested object or member element.

Attributes

Name	Type	Description
var	CDATA	Required - The variable name.
value	CDATA	Dynamic - The variable value. Alternatively, a value may be specified by a nested object or member element.
scope	local module application	The variable scope. Default scope is local (block visibility).

Body

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

`(object|member)?`

Related elements

<object>, <member>

2.2.1.5. The <use> Element**Purpose**

The <use> element applies a sequence of property settings and method invocations to an object.

The object to be used is referenced either by specifying the dynamic value attribute or by adding a nested <member> element as the first child.

The methods and properties are described by nested <property> and <method> elements.

Attributes

Name	Type	Description
<i>value</i>	CDATA	Dynamic - The value to be used. Evaluates to the object, to which method invocations and property settings are applied.

Body

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

`(member?, (method|property)+)`

Related elements

<member>, <property>, <method>

2.2.1.6. The <property> Element**Purpose**

The <property> element defines a name/value pair used to populate a Java Bean property or a `java.util.Map`. The property name is specified by the mandatory name attribute.

The property value may be specified either by the value attribute, a nested <object> or

`<member>` element.

A string 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. For other property types, the property value must be an instance of that type.

`<property>` elements may appear inside `<use>` and `<object>` elements.

Attributes

Name	Type	Description
<code>name</code>	NMTOKEN	Required - The name of the property (or association key).
<code>value</code>	CDATA	Dynamic - The property (or association) value.

Body

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

`(object|member)?`

Related elements

`<use>`, `<object>`, `<member>`

2.2.1.7. The `<object>` Element

Purpose

The `<object>` element creates a new object of the class specified by the `class` attribute.

If a `<constructor>` child element is present, it will be used to create the new object. Otherwise, `Class.newInstance()` will be called.

After instantiation, the object may be initialized by applying property settings and method invocations, described by nested `<property>` and `<method>` elements.

An `<object>` element may appear inside `<set>`, `<property>` and `<arg>` elements.

Attributes

Name	Type	Description
<code>class</code>	CDATA	Required, Dynamic - The class to be instantiated.

Body

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

(constructor?, (method|property)*)

Related elements

<set>, <property>, <arg>, <constructor>, <method>

2.2.1.8. The <constructor> Element

Purpose

The <constructor> element calls a constructor. Arguments may be given by nested <arg> elements.

A string argument value will be automatically converted to the formal constructor argument 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 argument is given in standard notation. For other argument types, the argument value must be an instance of that type.

A single <constructor> element may appear inside an <object> element.

Attributes

The <constructor> element has no attributes.

Body

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

(arg*)

Related elements

<object>

2.2.1.9. The <member> Element

Purpose

The <member> element is used to access an object- or class (static) member, which is either a method or a field.

The <member> element requires at least one of the `value` and `class` attributes. The `value` attribute evaluates to the object whose method or field is to be accessed. The `class` attribute specifies the class used to search for the member to be accessed. If both attributes are specified, the given value must be an instance of the given class. If the `class` attribute is omitted, the search class defaults to the given value's class. If the `value` attribute is omitted,

only static members of the given class can be accessed.

A `<member>` element may appear inside `<set>`, `<use>`, `<property>` and `<arg>` elements.

Attributes

Name	Type	Description
<code>class</code>	CDATA	Dynamic - The fully qualified class name used to search for method or field members.
<code>value</code>	CDATA	Dynamic - The value whose member is to be accessed.

Body

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

`(method|field)`

Related elements

`<set>`, `<use>`, `<property>`, `<method>`, `<field>`

2.2.1.10. The `<field>` Element

Purpose

The `<field>` element accesses a field member. The field name is specified by the mandatory name attribute.

A `<field>` element may appear inside `<member>` elements.

Attributes

Name	Type	Description
<code>name</code>	NMTOKEN	Required - The field name.

Body

The `<field>` element has no body.

`(property*)`

Related elements

`<member>`

2.2.1.11. The `<method>` Element

Purpose

The `<method>` element invokes a method. The method name is specified by the mandatory name attribute. Arguments may be given by nested `<arg>` elements.

A string argument value will be automatically converted to the formal method argument 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 argument is given in standard notation. For other argument types, the argument value must be an instance of that type.

`<method>` elements may appear inside `<use>` and `<object>` elements. A single `<method>` element may appear inside a `<member>` element.

Attributes

Name	Type	Description
name	NMTOKEN	Required - The method name.

Body

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

`(arg*)`

Related elements

`<use>`, `<object>`, `<member>`, `<arg>`

2.2.1.12. The `<arg>` Element**Purpose**

The `<arg>` element specifies a method or constructor argument.

The argument value may be specified either by the value attribute, a nested object or member element.

`<arg>` elements may appear inside `<method>` and `<constructor>` elements.

Attributes

Name	Type	Description
value	CDATA	Dynamic - The argument value. Alternatively, a value may be specified by a nested object or member element.

Body

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

```
(object|member)?
```

Related elements

```
<method>, <constructor>, <object>, <member>
```

2.2.2. Accessors

The `calyxo.base` accessors provides access to data related to the *Calyxo Base* 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`.

2.2.2.1. The `base.context` accessors

The `base.context.*` accessors supply application-global information.

In a string context, the expression `${calyxo.base.context}` itself can be used as a synonym for `${calyxo.base.context.path}`.

path

Answers the application context path by delegating to `HttpServletRequest.getContextPath()`.

Example

In the JSP fragment below, a resource is referenced by a context relative path without having to know the application context path.

```
<html>
  ...
  
  ...
</html>
```

```
</html>
```

home

Answers an absolute url pointing to the application root. The result is of the form `http://<host>:<port><context>`.

Example

Assuming an application running as `/foo` on host `www.calyxo.org` at port `8081`, the fragment

```
<html>
  <head>
    ...
    <base href="${calyxo.base.context.home}/index.jsp"/>
    ...
  </head>
  ...
</html>
```

will cause the browser to set the HTML base to `http://www.calyxo.org:8081/foo/index.jsp`.

name

Answers the display name of the servlet context by delegating to `ServletContext.getServletContextName()`.

2.2.2.2. The base.module accessors

Most of the `base.module.*` accessors delegate to the `ModuleContext` instance of the `module` they access. Usually, the accessed module is the current module. However, the `base.module.forName[...]` accessor can be used to get an accessor to a module other than the current.

In a string context, the expression `${calyxo.base.module}` itself can be used as a synonym for `${calyxo.base.module.name}`.

name

Answers the name of the accessed module by delegating to `ModuleContext.getName()`.

attribute[key]

Answers the value of a module scoped attribute for the given key by delegating to `ModuleContext.getAttribute(Object)`.

path[action]

Answers the context relative path for the given action by delegating to `ModuleContext.getPath(String)`.

The result will vary depending on the mapping for the accessed module. If it has been mapped using an extension mapping, the extension will be appended; if it has been mapped using a prefix mapping, the prefix will be prepended. The action may contain a query string (starting with '?') and anchor string (starting with '#').

Examples

The expression `${calyxo.base.module.path['/foo']}` evaluates to

- `/foo.do` if the current module has been mapped to extension `*.do`.
- `/cars/foo` if the current module has been mapped to prefix `/cars/*`.

The expression `${calyxo.base.module.path['/foo?bar=1#top']}` evaluates to

- `/foo.do?bar=1#top` if the current module has been mapped to extension `*.do`.
- `/cars/foo?bar=1#top` if the current module has been mapped to prefix `/cars/*`.

forName[name]

Answers a `base.module` accessor for the specified module by delegating to `ModuleSupport.getModuleContext(String)`.

If the given module does not exist, `null` is answered.

Example

The expression `${calyxo.base.module.forName['foo'].path['/bar']}` evaluates to the context relative path of action `/bar` in module `foo`.

2.2.2.3. The base.i18n accessors

The `base.i18n.*` accessors link to the `i18n` related services provided by the current module's `I18nSupport` instance.

locale

Answers the desired locale by delegating to `I18nSupport.getLocale(HttpServletRequest)`.

Example

The expression `${calyxo.base.i18n.locale.language}` evaluates to the language code for the request's desired locale.

format

Formats the specified message by delegating to `Message.format(HttpServletRequest,`

Locale, I18nSupport).

Example

The expression `${calyxo.base.i18n.format[message]}` formats the Message found at attribute "message".

bundle[name]

Answers an accessor, which may be used to retrieve localized resource strings and messages from the specified bundle.

- `resource[key]` – lookup resource string for the specified key; delegates to `I18nSupport.getResource(...)`
- `message[key][arg 1]...[arg n]` – lookup message template for the specified key and format it with the specified arguments; delegates to `I18nSupport.getMessage(...)`

Examples

Here are some expressions to illustrate looking up resources and messages:

- The expression `${calyxo.base.i18n.bundle['labels'].resource['hello']}` gets the resource for key "hello" in bundle "labels".
- The expression `${calyxo.base.i18n.bundle['messages'].message['welcome'][user]}` formats the one-argument message with key "welcome" in bundle "messages", using the result of expression `user` as argument.
- The expression `${calyxo.base.i18n.bundle['messages'].message['goodbye']}` formats the zero-argument message with key "goodbye" in bundle "messages".

If a bundle is used more than once in a page, it is recommended to create the bundle bean once, save it to page scope and reuse that instance in subsequent lookups:

```
<c:set var="labels" value="${calyxo.base.i18n.bundle['labels']}" />
...
${labels.resource['engine']}
...
${labels.resource['wheels']}
...
```

2.2.2.4. The base.eval accessor

eval[expression]

The `base.eval` accessor can be used to evaluate the specified EL expression string.

The expression string may refer to the implicit objects `param`, `requestScope`, `sessionScope`, `moduleScope` and `applicationScope`.

Example

The expression `${calyxo.base.eval[requestScope['role']]}` evaluates the string located at request scope attribute "role".

2.2.3. Functions

As covered in the [Configuration](#) section, *Calyxo* supports the use of EL expressions in attributes of configuration elements. Static methods of a class may be registered as EL functions with the `<functions>` element.

Standard Functions

The `de.odysseus.calyxo.base.misc.StandardFunctions` class provides the JSTL standard functions as defined in the [JSTL 1.1 specification](#).

The class has to be registered using the `<functions>` element as follows:

```
<base:functions
  prefix="fn"
  class="de.odysseus.calyxo.base.misc.StandardFunctions"/>
```

Example

The expression `${fn:split('foo/bar','/')}` evaluates to string array ["foo","bar"].

Module Functions

The `de.odysseus.calyxo.base.misc.ModuleFunctions` class provides module-related functions.

The class has to be registered using the `<functions>` element as follows:

```
<base:functions
  prefix="module"
  class="de.odysseus.calyxo.base.misc.ModuleFunctions"/>
```

Having this, the following functions are available in dynamic attribute expressions:

- `attribute(ModuleContext module, String name)` – Answer the value of the specified module scope attribute
- `initParameter(ModuleContext module, String name)` – Answer the value of the specified module init parameter
- `path(ModuleContext module, String action)` – Answer the context-related path for the specified action

Example

The expression `${module:attribute(moduleContext,'foo')}` evaluates to the module attribute at key "foo".

Type Functions

The `de.odysseus.calyxo.base.misc.TypeFunctions` class provides functions to convert between several Java types.

The class has to be registered using the `<functions>` element as follows:

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

Having this, the following functions are available in dynamic attribute expressions:

- `toByte(Object)` – convert to `java.lang.Byte` (accepts numbers and strings)
- `toShort(Object)` – convert to `java.lang.Short` (accepts numbers and strings)
- `toInteger(Object)` – convert to `java.lang.Integer` (accepts numbers and strings)
- `toLong(Object)` – convert to `java.lang.Long` (accepts numbers and strings)
- `toFloat(Object)` – convert to `java.lang.Float` (accepts numbers and strings)
- `toDouble(Object)` – convert to `java.lang.Double` (accepts numbers and strings)
- `toBigDecimal(Object)` – convert to `java.math.BigDecimal` (accepts numbers and strings)
- `toCharacter(Object)` – convert to `java.lang.Character` (accepts strings of length 1)
- `toString(Object)` – convert to `java.lang.String` (accepts any object)
- `toDate(Object)` – convert to `java.util.Date` (accepts strings)

Example

The expressions `${type:toInteger(123)}` and `${type:toInteger('123')}` both evaluate to a `java.lang.Integer` with value 123.

2.2.4. Tag Library

The *Calyxo Base* custom tag library contains tags to link or send a form to an action. In a JSP file, just associate the prefix you want to use for the tags with URI `http://calyx0.odysseus.de/jsp/base`:

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0"
  xmlns:base="http://calyx0.odysseus.de/jsp/base">
  ...
</jsp:root>
```

Since the tag library descriptor is contained in the *Calyxo Base* jar file, it is already available to applications. The container will automatically find it. For documentation purposes, a copy is located in `CALYX0_HOME/calyxo-base/conf/share/calyxo-base.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
<code>access</code>	Install the <i>Calyxo</i> accessors hierarchy in request scope.

HTML Tags

Name	Description
<code>a</code>	Render an HTML a tag that links to an action.
<code>form</code>	Render an HTML form tag to send form data to an action.

Attribute Groups

We use the following **abbreviations** to refer to groups of HTML attributes. They have the same meaning as in the HTML 4.01 specification.

Abbreviation	HTML Attributes
<code>%coreattrs</code>	<code>id</code> , <code>class</code> , <code>style</code> , <code>title</code>
<code>%i18n</code>	<code>lang</code> , <code>dir</code>
<code>%events</code>	<code>onclick</code> , <code>ondblclick</code> <code>onmousedown</code> , <code>onmouseup</code> , <code>onmouseover</code> , <code>onmousemove</code> , <code>onmouseout</code> <code>onkeydown</code> , <code>onkeyup</code> , <code>onkeypress</code>

All of them are mapped to the HTML attribute of the same name.

2.2.4.1. The <a> Tag

Purpose

The `<a>` tag defines an HTML link (i.e., it is mapped to an HTML `<a>` element).

The `module` and `action` attributes are used to address the action to be invoked by the link.

Attributes

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

Name	Description
<code>accesskey</code>	Mapped to the HTML attribute of the same name.
<code>action</code>	Required - Module-relative path to the action in the target module, that should be invoked when following the link. The tag will transform this into a context-relative path, when it renders the HTML form tag's action attribute. The action may be extended with a query string (starting with <code>?</code>) and an anchor (starting with <code>#</code>).

charset	Mapped to the HTML attribute of the same name.
coords	Mapped to the HTML attribute of the same name.
hreflang	Mapped to the HTML attribute of the same name.
module	The name of the module, that will be targeted by this link. The specified action will be searched in that module. If omitted, the link will target into the current module.
name	Mapped to the HTML attribute of the same name.
onblur	Mapped to the HTML attribute of the same name.
onfocus	Mapped to the HTML attribute of the same name.
rel	Mapped to the HTML attribute of the same name.
rev	Mapped to the HTML attribute of the same name.
shape	Mapped to the HTML attribute of the same name.
tabindex	Mapped to the HTML attribute of the same name.
target	Mapped to the HTML attribute of the same name.
type	Mapped to the HTML attribute of the same name.

Body

The <a> tag requires a body containing the content to be rendered as a link.

2.2.4.2. The <access> Tag**Purpose**

The <access> tag is used to install the accessor hierarchy in the specified request scope attribute.

Since the tag stores the accessors in request scope, it needs to be executed only once per request. E.g., if a template contains the tag, included templates do not need to also contain it.

Attributes

Name	Description
attribute	Required - Specifies the request scope attribute. As a general convention, the string value "calyxo" should be used.

Body

The <access> tag has no body.

2.2.4.3. The <form> Tag

Purpose

The <form> tag defines an HTML form (i.e., it is mapped to an HTML <form> element).

The action attribute is used to address the action to be invoked by the form.

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.
enctype	Mapped to the HTML attribute of the same name.
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.).

2.3. Extension Points

2.3.1. Access API

The access API classes are located in package `de.odysseus.calyxo.base.access`. The API allows application developers to implement their own accessors and add them to the hierarchy. The fundamental types are `Accessor` and `AccessorMap`. Roughly, these two keep the whole secret of how accessors work.

Accessors

The central interface is `Accessor`. It acts as a provider for objects, usually Java beans, Maps or

Collections. The

```
public Object get(HttpServletRequest request)
```

method will be called during expression evaluation for that purpose. The

```
public boolean isCacheable()
```

method should answer true iff the object returned by `get()` may be cached during a request. In other words, if the method doesn't use data which may change during the request. In this case, the `get()` method will be called only once per request.

Accessor Maps

An `AccessorMap` collects `Accessors` in a map. As such, it provides methods to put or remove accessors. Since it implements the `Accessor` interface itself, it may even contain nested `AccessorMaps`.

Its `get(HttpServletRequest)` method answers a map whose `get(Object)` method selects an `Accessor` by key and delegates to its `get(HttpServletRequest)` method.

Access Support

The `AccessSupport` class will be instantiated once per module and encapsulates the root `AccessorMap`. The class provides several static `getInstance(...)` methods to retrieve the module's instance. The `put(Object, Accessor)` method is used to add an accessor. The `create(HttpServletRequest)` method delegates to `get(HttpServletRequest)` of the wrapped `AccessorMap`.

Access Exceptions

The `Accessor`'s `get(HttpServletRequest)` method does not declare an `Exception`. This is because accessors usually delegate to other objects, especially to maps and since the `Map.get(Object)` method doesn't declare an exception either.

The API defines the runtime exception class `AccessException`, which should be thrown by accessors or by beans and maps returned by accessors.

Other Classes

The package contains a couple of other classes, which may serve as base classes for accessor implementations. Please refer to the API documentation for information on these.

Quick Example

To provide your own accessors, do the following:

1. Implement the `Accessor` interface.

```

public class FooAccessor implements Accessor {
    public Object get(HttpServletRequest request) {
        return "bar";
    }
    public boolean isCacheable() {
        return true;
    }
}

```

2. Implement a plugin (either *Calyxo Control* or Struts, depending on your environment). During initialization, add your accessor to the module's `AccessSupport` under a unique key. The following code puts an accessor map, containing our `FooAccessor` under key "foo", into key "sunshine":

```

ModuleContext moduleContext = ...
AccessSupport accessSupport = AccessSupport.getInstance(moduleContext);
AccessorMap accessors = new AccessorMap();
accessors.put("foo", new FooAccessor());
accessSupport.put("sunshine", accessors);

```

As an alternative (and probably much cooler), you could express the above in a configuration file with

```

<use>
  <member class="de.odysseus.calyxo.base.access.AccessSupport">
    <method name="getInstance">
      <arg value="{moduleContext}"/>
    </method>
  </member>
  <method name="put">
    <arg value="sunshine"/>
    <arg>
      <object class="de.odysseus.calyxo.base.access.AccessorMap">
        <method name="put">
          <arg value="foo"/>
          <arg>
            <object class="...FooAccessor"/>
          </arg>
        </method>
      </object>
    </arg>
  </method>
</use>

```

The `<base:access>` tag calls `AccessSupport.create(HttpServletRequest)` and stores the returned map into request scope. This map will delegate to your accessor's `get(HttpServletRequest)` method, when its key is requested as in the following code:

```

<base:access var="calyxo"/>
...
Let's go to the {calyxo.sunshine.foo}...

```

For more realistic code, see the various accessor implementations in package `de.odysseus.calyxo.base.misc`.

2.3.2. EL Functions

Since using EL expressions is a very handy feature for all kinds of purposes, they are supported by *Calyxo* in many places. As described in the [Functions Reference](#) section, *Calyxo* comes with three groups of predefined functions: Standard Functions (for string manipulation), Module Functions and Type Functions (for type conversions).

To be able to use additional functions in your EL expressions, you usually have to perform to steps:

1. Write a public Java class containing a public static method for every function you intend to use. The only restriction one has to obey is, that no multiple method names are used. That means that instead of implementing two methods `long max(long, long)` and `double max(double, double)` you have to choose different method names like `maxLong` and `maxDouble`, or similarly `long min(long, long)` and `long min(long, long, long)` might be renamed to `long min2(long, long)` and `long min3(long, long, long)`.
2. Register your Java class in that configuration file, where it shall be used (registered functions are not imported!). This is done by using the `<functions>` tag as described in the [Functions Reference](#) section.

Of course, the first step may be skipped if a suitable Java class does already exist, either written by yourself or taken from a third party library. In other cases, a class providing all required functions does exist, but is not suitable as a functions class because of multiple method names (examples are the classes `java.lang.Math`, `java.util.Arrays` and `java.util.Collections`, which supply a lot of useful methods for number and collection manipulations). Here it is sufficient to write a new class which delegates all functionality to the desired class.

Example

The following code illustrates the skeleton of a class providing some useful functions for date calculations.

```
package my.functions;

public class DateFunctions {
    public static Date today() {
        // calculate the current date
        return date;
    }
    public static boolean isFuture(Date date) {
        // test if date is larger than today
        return result;
    }
    public static boolean isPast(Date date) {
```

```
// test if date is smaller than today
return result;
}
public static long daysBetween(Date begin, Date end) {
    // calculate difference between the two dates
    return days;
}
}
```

The class is registered in your configuration file by writing

```
<base:functions prefix="date" class="my.functions.DateFunctions"/>
```

Now the above functions may be accessed like follows: `${date:daysBetween(date:today(), departureDate)}`.

2.4. Integration

When using the *Calyxo* controller, there's nothing to do: the *Calyxo Base* component is a mandatory part of the *Calyxo* platform and is always available.

In order to use the *Calyxo Base* component with Struts, it must be somehow integrated into the application's controller. This is achieved by the *Calyxo Base* plugin for Struts. See the [Calyxo Struts](#) component for instructions on loading the plugin.

3. Calyxo Control

The controller lies at the heart of a JSP Model 2 application. The *Calyxo Control* component provides a clean, module-aware implementation realizing the *Front/Application Controller*, *Service To Worker* and *Intercepting Filter* design patterns.

Calyxo Control versus Struts

Calyxo Control uses an approach similar to Struts, so Struts users should feel familiar with *Calyxo Control* right from the start. However, there are some major differences.

We won't make a detailed comparison to the Struts controller. It should be stated, that *Calyxo Control* has been designed as a relative to the Struts controller, while fixing some Struts design issues and adding some important new features:

- *Calyxo Control* supports action *filters*. A filter *chain* may be attached to an action in its configuration. Action filters realize the *Interception Filter* design pattern.
- *Calyxo Control* doesn't mix validation issues into the core configuration like Struts does. Instead, form validation is incorporated as an action filter.
- *Calyxo Control* action commands implement an action *interface* rather than having to subclass from a base action class.
- *Calyxo Control* supports the concept of *dispatchers*. A dispatcher is responsible for delegating to a view according to an action's result. E.g., you may use custom dispatchers to render different content types.
- *Calyxo Control* uses one servlet per module. This way, selecting the module for an incoming request is left to the servlet container.

If you still want (or need) to use Struts as your controller, please refer to the [Calyxo Struts](#) component.

3.1. Calyxo Control Concepts

Calyxo has been designed to ease the development of JSP Model 2 applications. In a Model 2 application servlets take over the controller part and JSPs are used as view technology.

The *Calyxo* controller realizes the *Command and Controller* strategy. This strategy builds on the *Front Controller* and *Application Controller* design patterns. The Front Controller, implemented as a servlet, acts as a centralized access point for incoming requests. It delegates to an Application Controller (*module*), which is responsible for identifying and invoking *Command Objects (actions)* and for identifying and dispatching to views.

Note

Please refer to the [Calyxo Base Concepts](#) section as a prerequisite to fully understand *Calyxo Control*.

Let's introduce the major participants in the controller game.

Module Servlets

Each module is associated with its own *module servlet*. For an incoming request, the current module is selected by the servlet container. According to the servlet's url mapping a module may be selected by either a prefix- or extension mapping.

The servlet then extracts the module-relative action path from the request URL and invokes the current module's `process()` method, passing over the request, response and action path.

Modules

The module digs for an *action* to be invoked for the requested action path. The action's *command*, consisting of a sequence of *filters* and a core action command implementation, is then executed by the module.

A command's execution results in a *dispatch* configuration, which states how to finish the action processing cycle. This final action processing step is performed by a *dispatcher*. E.g., a dispatcher might delegate to another action (optionally within another module) or to a context-relative resource path (like a JSP).

Actions

A module is a container for *actions*. Each action is associated with a module-relative path. Depending on the incoming request, the module selects and *invokes* an action.

An *action* by itself may consist of a core action command implementation, a set of parameters, a chain of *filters*, *exception handlers* and a *dispatcher*.

Providing action command implementations is your part. They are the controller's end, from where you may invoke your application specific business logic. Action command implementations provide the core processing of an action.

Filters

A series of filters may be attached to an action. The module combines these into a chain, appending the core action command as the last link. The chain presents itself as a command, which is initially executed by the module.

A Filter intercepts the chain's execution. As a link in the chain it executes the remaining chain and may perform preprocessing, postprocessing or both. It may even decide to not execute the remaining chain.

Exception Handlers

If an action's invocation throws an exception, the module searches for an exception handler and - if it finds one - executes it. Exception handlers may be declared for a hierarchy of exception classes either local to an action or global to the module. Local handlers precede over global handlers.

Dispatchers

An action invocation results in information on how to *dispatch*. Dispatching is the final step in action processing. This might result in delegating to another action (optionally within another module), to a JSP or to an external location. Dispatching might use a forward, include or redirect.

The default dispatcher does all this. However, in certain cases special treatment may be desired. Therefore, customized dispatchers can be implemented and used.

Plugins

Plugins provide a way to extend the controller by executing code during module initialization and finalization. The Plugin API is used to register filter classes by name and to bring custom dispatchers into the system.

3.1.1. Modules

For an incoming request, the module servlet selects its associated module to be the current module for this request and delegates action processing to it. The module selects an action, executes it and dispatches according to the action's result.

The term module is used as a synonym for *module controller*.

Module Context

Here are some details of the module context implementation used by the *Calyxo Control* component:

- `getName()` - answers the module's servlet name.
- `getInitParameter(String name)` - answers the module's servlet configuration init parameter for the given name.
- `getPath(String action)` - answers a context-relative path for the specified action path, according to the module's servlet mapping. The action may have a query and/or anchor appended.
- `getClassLoader()` - answers the module class loader that has been set either as a constructor parameter or via the `setClassLoader` method. If no class loader has been set, answers `getClass().getClassLoader()`.
- Module scope is simulated by maintaining a hash map.

Module Initialization

A module servlet creates its module when it gets initialized by the servlet container. The module then starts to initialize itself in several steps:

1. Create and install the module's i18n support - The i18n support class may be given by module init parameter i18n-support. If it is omitted, `de.odysseus.calyxo.control.impl.DefaultI18nSupport` is taken as default, which uses Java's resource bundle mechanism.
2. Create and install the module's message support - The message support class may be given by module init parameter message-support. If it is omitted, `de.odysseus.calyxo.control.impl.DefaultMessageSupport` is taken as default, which saves messages into session scope.
3. Create and install the `calyxo.base` and `calyxo.control` accessors.
4. Parse the control configuration file - the context-relative path to the configuration file is taken from module init parameter `config`. The parse results in a tree of objects corresponding to the configuration elements.
5. Load plugins - Instantiate and initialize `Plugin` objects.
6. Create and initialize actions - Instantiate and initialize `Action` and `Filter` objects and combine them into a command chain.
7. Create and initialize exception handlers - Instantiate and initialize `ExceptionHandler` objects.

Once initialized, the module is ready for action processing.

Action Processing

For some module-relative path, action processing involves the following steps:

1. Lookup the action for the requested path. If there's no action for that path, try path `"/*`". If there's neither action, throw an exception.
2. Invoke the action by executing its command chain.
3. If the action throws an exception, search for an exception handler and - if one could be found - invoke it; otherwise, throw the exception again.
4. If the action returned a dispatch configuration, select a dispatcher and invoke it.

3.1.1.1. Module Servlets

As already mentioned, a module takes its own servlet. Module servlets are defined in the web application's deployment descriptor, which has to be made available as `/WEB-INF/web.xml`.

For each module, you have to

1. define a module servlet; the module name is given by the servlet name.
2. pass the context-relative path to the configuration file as initialization parameter `config`.
3. define exactly one servlet mapping for the module; a module may be mapped either using
 - an *extension mapping* - this will map request URLs, whose context-relative paths end with some `".ext"`, to the module - or
 - a *prefix mapping* to map request URLs, whose context-relative paths start with some `"/prefix"`, to the module.

Here's an example web.xml, which uses the modules *foo* and *bar*. We'll map prefix */foo* to module *foo* and extension **.bar* to module *bar*:

```
<web-app xmlns="http://java.sun.com/xml/ns/j2ee" version="2.4">
  ...
  <servlet>
    <servlet-name>foo</servlet-name>
    <servlet-class>de.odysseus.calyxo.control.ModuleServlet</servlet-class>
    <init-param>
      <param-name>config</param-name>
      <param-value>/WEB-INF/foo/calyxo-control-config.xml</param-value>
    </init-param>
    <load-on-startup/>
  </servlet>

  <servlet>
    <servlet-name>bar</servlet-name>
    <servlet-class>de.odysseus.calyxo.control.ModuleServlet</servlet-class>
    <init-param>
      <param-name>config</param-name>
      <param-value>/WEB-INF/bar/calyxo-control-config.xml</param-value>
    </init-param>
    <load-on-startup/>
  </servlet>

  <servlet-mapping>
    <servlet-name>foo</servlet-name>
    <url-pattern>/foo/*</url-pattern>
  </servlet-mapping>

  <servlet-mapping>
    <servlet-name>bar</servlet-name>
    <url-pattern>*.bar</url-pattern>
  </servlet-mapping>
  ...
</web-app>
```

The optional `load-on-startup` element indicates that this servlet should be loaded on the startup of the web application.

Now, if this application gets deployed under context `/sample` on `http://localhost:8080`, then

- the request URL `http://localhost:8080/sample/foo/reach?what=stars` will select module *foo* with action path `/reach`,
- and request URL `http://localhost:8080/sample/walk.bar?where=moon` will select module *bar* with action path `/walk`.

It is probably bad practice to have both, prefix-mapped modules and extension-mapped modules, within one application (as in our example), because this may easily cause mapping conflicts to occur.

Just consider the URL `http://localhost:8080/sample/foo/clash.bar...` which module servlet

should be selected by the container?

3.1.2. Plugins

Plugins implement the `de.odysseus.calyxo.control.Plugin` interface. When the module gets loaded, it will instantiate the declared plugins and call their

```
public void init(
    PluginConfig config,
    PluginContext context) throws Exception;
```

method. The `de.odysseus.calyxo.control.conf.PluginConfig` interface reflects the plugin's configuration and provides methods to access its parameters.

The `de.odysseus.calyxo.control.PluginContext` class has methods

- `setFilterClass(String name, Class clazz)` to register a filter class under some name. A registered filter class can be referenced by name using `<filter name="...">` declarations. See the section on [filter configurations](#) for details.
- `setDispatcher(String name, Dispatcher dispatcher)` to register a dispatcher under some name. Custom dispatchers must be registered before they can be referenced using `<action dispatcher="...">` and `<dispatch dispatcher="...">` declarations.
- `setDefaultDispatcher(Dispatcher dispatcher)` to set the default dispatcher.
- `setDefaultActionClass(Class clazz)` to set the default action class. The module will use this class if an `<action>` element does not specify the `class` attribute.

Furthermore, the `getModuleContext()` method is used to access the module context.

3.1.3. Actions

An action describes how to process requests with a particular module-relative path. An action is made of

- a set of [parameter configurations](#),
- a sequence of [filters](#),
- a core [action command](#),
- a set of [dispatch configurations](#),
- a set of [exception handlers](#),
- an optional [dispatcher](#).

Filters, action commands and exception handlers are instantiated and initialized by the module during startup. There will be exactly one instance per corresponding configuration element.

The controller combines an action's filters and command into a chain, the action command being the last link. Such a chain presents itself as a `Command` implementation. "Invoking an action" is a synonym for executing the chain's `execute()` method.

An action invocation results in a dispatch configuration. Typically, an action command selects one of its dispatch configurations. However, an action may also dynamically create a dispatch

result.

If a filter or action command throws an exception, the module will catch it, select an exception handler and invoke it. In this case, the dispatch configuration will be provided by the exception handler.

The resulting dispatch configuration is then passed to the dispatcher to dispatch the request.

3.1.3.1. Action Commands

At the very basic, the `de.odysseus.calyxo.control.Command` interface represents something, that can be executed taking a http servlet request and response as parameters. The result of a command execution contains information on how to dispatch the request. To be more concrete, the `Command` interface defines the single method

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

The `de.odysseus.calyxo.control.conf.DispatchConfig` interface reflects a [dispatch configuration](#), which contains information on how to dispatch the request.

Though uncommon, a command may generate and write the response on its own, making dispatching obsolete. A command indicates this by answering `null`.

The `Command` interface is not intended to be directly implemented by application developers. However, application developers come into touch with it at two points:

First, the controller combines an action's filters and command into a chain of `Commands`. Therefore, the remaining chain is passed to a filter as a `Command` object.

And second, action commands have to implement the `de.odysseus.calyxo.control.Action` interface, which extends the `Command` interface by adding method

```
public void init(
    ActionConfig config,
    ModuleContext context) throws Exception;
```

After instantiation, the controller gives an action the opportunity to do some initialization by calling its `init()` method. The `de.odysseus.calyxo.control.conf.ActionConfig` interface reflects the action's configuration and provides methods to access parameters, find dispatch configurations and more.

The `execute()` method will be called during action invocation, as the action command is the last link in the command chain executed by the controller.

The controller instantiates exactly one instance per action configuration. As an important consequence, an action command can maintain its configuration as its internal state. On the other hand, it has to be thread safe, since parallel invocations of the same action will execute the same action command instance.

Since Action is an interface, developers are free to implement action commands anywhere in their class hierarchy. However, *Calyxo Control* supplies a convenient base action command class, namely `de.odysseus.calyxo.control.misc.AbstractAction`. If you do not need to inherit from any particular class, we recommend using this one. It provides access to the action configuration, module context and message support objects. Its implementation of `init(ActionConfig, ModuleContext)` delegates to an `init()` method, which may be implemented by subclasses to do further initialization.

Consider the following simple action configuration:

```
<action path="/login" class="org.foo.bar.LoginAction">
  <dispatch name="success" action="/welcome"/>
  <dispatch name="failure" path="/WEB-INF/jsp/login.jsp"/>
</action>
```

The `/login` action defines class `org.foo.bar.LoginAction` to be its action command class.

Directly implementing the Action interface

A direct Action implementation might look like this:

```
package org.foo.bar;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import de.odysseus.calyxo.base.Message;
import de.odysseus.calyxo.base.ModuleContext;
import de.odysseus.calyxo.control.Action;
import de.odysseus.calyxo.control.MessageSupport;
import de.odysseus.calyxo.control.conf.ActionConfig;
import de.odysseus.calyxo.control.conf.DispatchConfig;

public class LoginAction implements Action {
    private ActionConfig config;
    private MessageSupport messages;

    /**
     * Initialize action.
     */
    public void init(ActionConfig config, ModuleContext context)
        throws Exception {
        this.config = config;
        this.messages = MessageSupport.getInstance(context);
    }

    /**
     * Execute action.
     */
    public DispatchConfig execute(
```

```

HttpServletRequest request,
HttpServletRequest response) throws Exception {

    boolean ok = false;
    // do login...

    if (ok) {
        return config.findDispatchConfig("success");
    } else {
        Message message = new Message("messages", "login.failed");
        messages.addError(request, message);
        return config.findDispatchConfig("failure");
    }
}
}
}

```

Inheriting from AbstractAction

Let's rewrite our action by making it a subclass of AbstractAction and see what it looks like:

```

package org.foo.bar;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import de.odysseus.calyxo.base.Message;
import de.odysseus.calyxo.control.conf.DispatchConfig;
import de.odysseus.calyxo.control.misc.AbstractAction;

public class LoginAction extends AbstractAction {
    /**
     * Execute action.
     */
    public DispatchConfig execute(
        HttpServletRequest request,
        HttpServletResponse response) throws Exception {

        boolean ok = false;
        // do login...

        if (ok) {
            return getActionConfig().findDispatchConfig("success");
        } else {
            Message message = new Message("messages", "login.failed");
            getMessageSupport().addError(request, message);
            return getActionConfig().findDispatchConfig("failure");
        }
    }
}
}

```

Quite compact, isn't it?

3.1.3.2. Action Filters

Filters implement the `de.odysseus.calyxo.control.Filter` interface. As with action commands, the controller instantiates one instance per filter configuration. After instantiation, the controller gives a filter the opportunity to do some initialization by calling its

```
public void init(
    FilterConfig config,
    ModuleContext context) throws Exception;
```

method. The `de.odysseus.calyxo.control.conf.FilterConfig` interface reflects the filter's configuration and provides methods to access its parameters as well as the enclosing action configuration.

During action invocation, the

```
public DispatchConfig filter(
    HttpServletRequest request,
    HttpServletResponse response,
    Command command) throws Exception;
```

method will be called, as the filter is a link in the chain executed by the controller. The remaining chain is represented by the command parameter.

Filters may be configured with a JSP EL expression to be conditionally included into the chain. See the section on [Filter Configuration](#) for more on this.

Generally, command filtering can be divided into three optional steps:

1. command preprocessing,
2. command execution,
3. command postprocessing.

Action filters may use the same techniques as servlet filters, including wrapping the request and/or the response.

A filter may decide to *not* execute the remaining chain. For example, if a filter doing user input validations detected invalid fields, it would want to answer some dispatch configuration to redisplay the last page, rather than continuing with command execution.

Finally, an action filter may inspect (and even replace) the result of the command execution.

3.1.4. Exception Handlers

Exception handlers implement the `de.odysseus.calyxo.control.Filter` interface. The controller instantiates one instance per exception handler configuration. After instantiation, the controller gives an exception handler the opportunity to do some initialization by calling its

```
public void init(
    ExceptionHandlerConfig config,
    ModuleContext context) throws Exception;
```

method. The `de.odysseus.calyxo.control.conf.ExceptionHandlerConfig` interface reflects the handler's configuration and provides methods to access its parameters.

Exception handlers may be declared for a hierarchy of exception classes either local to an action or global to the module. See the section on [exception handler configuration](#) for details.

If an action's invocation throws an exception, the module searches for an exception handler as follows: for each class `cls` on the inheritance path from the exception's class up to `java.lang.Exception`, the module first checks for a local, then for or global exception handler, that has been declared for the current class `cls`. On the first one it finds, it calls the

```
public DispatchConfig handle(
    HttpServletRequest request,
    HttpServletResponse response,
    ActionConfig actionConfig,
    Exception exception) throws IOException, ServletException;
```

method. The `actionConfig` parameter corresponds to the action, that threw the exception. The handler is responsible to answer a dispatch configuration.

If the module didn't find a matching exception handler it wraps the exception into a `ServletException` and throws it again. Similar, an exception, thrown in an exception handler's `handler()` method is *not* caught by the module and will go up to the servlet container.

3.1.5. Dispatchers

Dispatchers implement the `de.odysseus.calyxo.control.Dispatcher` interface, which defines the single method

```
public void dispatch(
    HttpServletRequest request,
    HttpServletResponse response,
    DispatchConfig config) throws IOException, ServletException;
```

The `de.odysseus.calyxo.control.conf.DispatchConfig` interface reflects a [dispatch configuration](#) and provides methods to access the dispatch path, module and action, the redirection flag as well as the dispatch parameters.

A custom dispatcher has to be registered by a plugin using either method of `de.odysseus.calyxo.control.PluginContext`:

- `setDispatcher(String, Dispatcher)` - to register a dispatcher under some name
- `setDefaultDispatcher(Dispatcher)` - to set the default dispatcher.

Custom dispatchers may be referenced by name in `<action>` and `<dispatch>` configurations through their `dispatcher` attribute.

An action invocation results in a dispatch configuration. The module selects a dispatcher as follows: if the dispatch configuration has a dispatcher set, use it; otherwise, if the action configuration has a dispatcher set, use it; otherwise, use the default dispatcher. In other words, the dispatch's dispatcher is most significant, followed by the action's dispatcher, followed by the default dispatcher.

If not overridden, *Calyxo Control* uses `de.odysseus.calyxo.control.impl.DefaultDispatcher` as the default dispatcher. This dispatcher processes a dispatch configuration as follows:

1. if the dispatch configuration specifies an action (and a module),
path := context-relative path of that action (in that module)
2. if the dispatch configuration specifies a path,
path := that path
3. append the dispatch parameters to *path* as HTTP query parameters
4. if the redirection flag is set and *path* starts with '/',
prepend the context path to *path*
5. dispatch using redirect, forward or include:
 - if the redirection flag is set, redirect the response to *path*
 - otherwise, if the response is not yet committed, forward to *path*
 - otherwise, include *path*

Custom dispatchers are free to reuse the default dispatcher by subclassing it or delegating to it.

3.1.6. Configuration

As usual, the *Calyxo Control* 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://calyx0.odysseus.de/xml/ns/control`.

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 Control* document type:

```
<!DOCTYPE calyx0-control-config
  PUBLIC "-//Odysseus Software GmbH//DTD Calyx0 Control 0.9//EN"
  "calyx0-control-config.dtd">
```

Copies of the DTD and XSD are located at

CALYXO_HOME/calyxo-control/conf/share/calyxo-control-config.*.

Document structure

The root element is `<calyxo-control-config>`. 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 schema/DTD version.

The root 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 these elements 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 optionally contains

1. a `<plugins>` element containing `<plugin>` elements,
2. a `<dispatches>` element, containing global `<dispatch>` elements,
3. an `<exception-handlers>` element, containing global `<exception-handler>` elements,
4. an `<actions>` element, containing `<action>` elements.

Thus, our configuration document has a general structure like this:

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

  <plugins>
    <!-- plugin definitions go here -->
  </plugins>

  <dispatches>
    <!-- dispatch definitions go here -->
  </dispatches>

  <exception-handlers>
    <!-- exception handler definitions go here -->
  </exception-handlers>

  <actions>
    <!-- action definitions go here -->
  </actions>

</calyxo-control-config>
```

Each of the nested elements is optional and thus may be omitted.

Configuration Interfaces

All configuration elements have corresponding Java interfaces. When actions, filters or exception handlers are initialized by the controller, an instance reflecting their corresponding configuration element is passed to them.

The configuration interfaces are located in package `de.odysseus.calyxo.control.conf`.

3.1.6.1. Parameter Configurations

Each of the `<plugin>`, `<action>`, `<filter>`, `<dispatch>` and `<exception-handler>` elements may be configured to take a set of parameters by specifying nested `<param>` elements.

Consequently, the `PluginConfig`, `ActionConfig`, `FilterConfig`, `DispatchConfig` and `ExceptionConfig` interfaces extend the `ParamsConfig` interface, which provides the `getParamConfig(String)` method to lookup a parameter configuration by its name.

A `<param>` element itself requires the two attributes

- `name`, specifying the parameter name, and
- `value`, specifying the parameter value.

The `value` attribute is dynamic (that is, its value may contain expressions) and of type string.

A parameter configuration is reflected by the `ParamConfig` interface, which defines the `getName()` and `getValue()` methods.

Example

The following action element defines parameter `users`.

```
<action path="/login" class="org.foo.bar.LoginAction">
  <param name="users" value="/WEB-INF/users.xml"/>
  <dispatch name="success" action="/welcome"/>
  <dispatch name="failure" path="/WEB-INF/jsp/login.jsp"/>
</action>
```

Now, the `org.foo.bar.LoginAction` action implementation might access the corresponding `ParamConfig` like this:

```
import de.odysseus.calyxo.control.conf.ParamConfig;
import de.odysseus.calyxo.base.conf.ConfigException;

public class LoginAction extends de.odysseus.calyxo.control.misc.AbstractAction {
  private String users;

  public void init() throws ConfigException {
    ParamConfig param = getActionConfig().getParamConfig("users");
    if (param == null) {
      throw new ConfigException("Missing parameter 'users'!");
    }
  }
}
```

```

    }
    users = param.getValue();
  }
  ...
}

```

3.1.6.2. Plugin Configurations

A plugin is defined using the `<plugin>` element, which may appear inside the `<plugins>` element. It takes a mandatory `class` attribute, giving the name of a class implementing the `Plugin` interface.

The `<plugin>` element may contain any number of `<param>` elements.

A plugin configuration is reflected by the `PluginConfig` interface, which defines the `getParamConfig()` method to lookup parameters by name.

Example

Here's a plugin configuration for plugin class `org.foo.bar.FooPlugin`, passing parameter `bar`.

```

<plugin class="org.foo.bar.FooPlugin">
  <param name="bar" value="foobar"/>
</action>

```

The controller will instantiate an instance of `FooPlugin` during module load and call its

```

public void init(
    PluginConfig config,
    PluginContext context) throws Exception;

```

method. The `PluginContext` gains access to the module context and contains methods to register dispatchers and filter classes by name. See the general [plugins](#) section for details.

3.1.6.3. Filter Configurations

Filters are added to an action using the `<filter>` element. They appear as children of an `<action>` element, following the action's `<param>` elements.

The `class` attribute may be used to identify a class implementing the `Filter` interface. As an alternative, the `name` attribute may be used to reference a filter class, that has been registered by a plugin under some unique name.

The `when` attribute can be used to specify a JSP EL expression (without `"${"` and `"}"`) to conditionally execute the filter. During action invocation, the expression will be evaluated to a boolean value. If it results to `true`, the filter is executed as usual. Otherwise, the filter will be skipped and is not part of the filter chain for that request.

The `<filter>` element may contain any number of `<param>` elements as well as `<dispatch>`

elements.

The Filter interface defines the `init()` and `filter()` methods. For an action, the controller instantiates the filter instances and arranges them into a sequence in the same order the `<filter>` elements appear within the `<action>` element.

Since a filter class may be registered by a plugin, it is not uncommon for filters to implement both, the `Plugin` and `Filter` interfaces.

Examples

Calyxo Control comes with some ready-to-use filters, that we'll use here to illustrate filter definitions.

Branch Filter

The *branch* simply dispatches to a target according to its configuration. It is used with a filter when condition. The filter may be attached to an action like this:

```
<action path="/login" class="org.foo.bar.LoginAction">
  <filter class="de.odysseus.calyxo.control.misc.BranchFilter"
    when="!empty param.forgotten">
    <dispatch action="/recoverPassword"/>
  </filter>
  ...
</action>
```

The filter also implements the `Plugin` interface to associate the filter with some name. The default filter name is "branch". Therefore, to reference the filter by its default name, we add the plugin configuration

```
<plugin class="de.odysseus.calyxo.control.misc.BranchFilter"/>
```

to the `<plugins>` element. Now, we can rewrite the above action definition as

```
<action path="/login" class="org.foo.bar.LoginAction">
  <filter name="branch" when="!empty param.forgotten">
    <dispatch action="/recoverPassword"/>
  </filter>
  ...
</action>
```

The dispatch configuration may be either specified directly by a nested anonymous dispatch configuration (as above) or by the target parameter referencing a dispatch configuration visible to the enclosing action:

```
<action path="/login" class="org.foo.bar.LoginAction">
  <filter name="branch" when="!empty param.forgotten">
    <param name="target" value="recover"/>
  </filter>
```

```

...
<dispatch name="recover" action="/recoverPassword"/>
</action>

```

No-Cache Filter

The *no-cache* filter adds some HTTP response headers to prevent browsers from caching the page. The filter may be attached to an action like this:

```

<action path="/login" class="org.foo.bar.LoginAction">
  <filter class="de.odysseus.calyxo.control.misc.NoCacheFilter"/>
  ...
</action>

```

The filter also implements the `Plugin` interface to associate the filter with some name. The default filter name is "no-cache". Therefore, to reference the filter by its default name, we add the plugin configuration

```
<plugin class="de.odysseus.calyxo.control.misc.NoCacheFilter"/>
```

to the `<plugins>` element. Now, we can rewrite the above action definition as

```

<action path="/login" class="org.foo.bar.LoginAction">
  <filter name="no-cache"/>
  ...
</action>

```

Cancel Filter

Consider a form containing a "Cancel" button. When the user presses this button, we usually do not want the core action implementation to handle this. The *cancel* filter checks for the existence of a request parameter, "cancel" by default, and - if it exists - directly dispatches to some target, without executing the remaining chain. Again, the default dispatch target name is "cancel".

This could be done with the branch filter. However, since this is a common task, the filter is provided for convenience.

```

<action path="/register" class="org.foo.bar.RegisterAction">
  <filter class="de.odysseus.calyxo.control.misc.CancelFilter"/>
  ...
  <dispatch name="cancel" action="/goodbye"/>
</action>

```

Above, we used the filter's default "cancel" for the request parameter name and target dispatch name. If the request parameter name were "abort" and we wanted to use a global dispatch configuration named "goodbye", our action might look like this:

```

<action path="/register" class="org.foo.bar.RegisterAction">
  <filter class="de.odysseus.calyxo.control.misc.CancelFilter">

```

```

    <param name="parameter" value="abort"/>
    <param name="target" value="goodbye"/>
  </filter>
  ...
</action>

```

The `CancelFilter` class also implements the `Plugin` interface to associate the filter with some name. The default filter name is - no wonder - "cancel". Thus, after adding `CancelFilter` as a plugin, we could rewrite the above examples by replacing `class="..."` with `name="cancel"`.

Finally, the target dispatch configuration may also be specified directly using a nested anonymous dispatch configuration element:

```

<action path="/register" class="org.foo.bar.RegisterAction">
  <filter name="cancel">
    <dispatch action="/goodbye"/>
  </filter>
  ...
</action>

```

3.1.6.4. Dispatch Configurations

The `<dispatch>` element is used to define a dispatch configuration. It may appear either inside the `<dispatches>` element to define a *global* dispatch configuration or inside an `<action>` element to define a *local* dispatch configuration.

A dispatch configuration contains information about how to proceed after action invocation. It may either point to another action or to a resource path. It may also specify a custom dispatcher and whether to use redirection.

The `<dispatch>` element takes the following attributes:

- The name attribute - specifying the name of the dispatch configuration. Actions use this name to lookup a configuration in the pool of local and global dispatch configurations. Omitting the name attribute creates a "default" dispatch configuration whose name property is null.
- The action attribute, optionally in conjunction with the module attribute - to dispatch to another action; if the module attribute is omitted, the current module is assumed.
- The path attribute - to dispatch to that path; e.g., the path may be a context-relative. However, custom dispatchers may interpret this attribute in their own fashion.
- The redirect attribute - specifying to redirect the response, if set to true.
- The dispatcher attribute - specifying a custom dispatcher to be used; custom dispatchers are registered by plugins.

It is an error to specifying both, the action *and* path attributes. Furthermore, the module attribute is only permitted in conjunction with the action attribute.

The body of the `<action>` element may contain a set of `<param>` elements. Dispatchers may use parameters in their own way. However, the default dispatcher will append them as request parameters.

A dispatch configuration is reflected by the `DispatchConfig` interface, which defines methods to access the attribute values as well as the `getParamConfig()` method to lookup parameters by name.

The `DispatchConfig` interface is the result type of the action's `execute()` method. An action uses its action configuration, `ActionConfig`, to lookup a dispatch configuration by name, using the

```
public DispatchConfig findDispatchConfig(String name);
```

method. The default dispatch configuration can be accessed by specifying name `null`.

Examples

Dispatching to an action in the current module

```
<dispatch name="foo" action="/bar"/>
```

dispatches to action with `path="/bar"` in the current module.

Dispatching to an action in another module

```
<dispatch name="foo" module="other" action="/bar"/>
```

dispatches to action with `path="/bar"` in the module `other`.

Dispatching to a context-relative path

```
<dispatch name="foo" path="/WEB-INF/jsp/bar.jsp"/>
```

dispatches to JSP at `/WEB-INF/jsp/bar.jsp`.

Adding parameters

```
<dispatch name="foo" path="/WEB-INF/jsp/bar.jsp">
  <param name="mode" value="lazy"/>
</dispatch>
```

dispatches to JSP using request path `/WEB-INF/jsp/bar.jsp?mode=lazy`.

Using redirection

```
<dispatch name="foo" path="/jsp/bar.jsp" redirect="true"/>
```

redirects to JSP at context-relative path `/jsp/bar.jsp`, whereas

```
<dispatch name="foo" path="http://calyxo.org" redirect="true"/>
```

redirects to absolute URL `http://calyxo.org`.

Using a custom dispatcher

Consider you implemented a custom dispatcher, which wraps the response to capture and buffer its content, before dispatching to a resource producing some XML. Then, it gets the the xml content and and applies an XSL transformation on it. Say, you registered this accessor as "xslt" using a plugin. After all,

```
<dispatch name="foo" dispatcher="xslt" path="/bar.xml">
  <param name="stylesheet" value="/WEB-INF/xsl/style.xsl"/>
</dispatch/>
```

will cause your custom dispatcher to apply XSLT stylesheet /WEB-INF/xsl/style.xsl to the content of /bar.xml.

3.1.6.5. Exception Configurations

An exception handler is defined using the `<exception-handler>` element. It may appear either inside the `<exception-handlers>` element to define a *global* exception handler or inside an `<action>` element to define a *local* exception handler.

The `<exception-handler>` element takes a mandatory `class` attribute, giving the name of the exception handler class, which has to implement the `ExceptionHandler` interface. The optional `type` attribute specifies the name of the exception type, that will be treated by the handler. If omitted, a `java.lang.Exception` will be assumed.

The `<exception-handler>` element may contain any number of `<param>` and `<dispatch>` elements.

An exception configuration is reflected by the `ExceptionHandlerConfig` interface, which defines the `getParamConfig()` method to lookup parameters by name.

Exception handlers are instantiated and initialized when the module gets loaded, one for each configuration. If an action invocation throws an exception, the module catches it and searches for an exception handler configuration as follows: for each class *cls* on the inheritance path from the exception's class up to `java.lang.Exception`, it first checks for a local, then for or global exception configuration whose `type` attribute matches the current class *cls*. If it finds one, it gets the corresponding `ExceptionHandler` instance and invokes its `handle()` method. Otherwise, the exception is wrapped into a `ServletException` and thrown again.

Examples

The following examples show some exception handler configurations. Whether a handler is selected for a concrete exception, is determined by the module's selection algorithm as explained above and also depends on whether it is defined as a local or global exception handler.

Using the simple exception handler

Calyxo Control supplies a simple exception handler implementation. This generic exception handler requires the parameters `bundle` and `key` to identify a message template. It will create and save a message object into the current request, passing the exception message as an argument. Finally, it looks up and returns a dispatch configuration in the action identified by the target parameter.

```
<exception-handler
  class="de.odysseus.calyxo.control.misc.SimpleExceptionHandler">
  <param name="bundle" value="messages"/>
  <param name="key" value="exception"/>
  <dispatch path="/WEB-INF/jsp/exception.jsp"/>
</exception-handler>
```

The dispatch configuration may be either specified directly by a nested anonymous dispatch configuration (as above) or by the target parameter referencing a dispatch configuration visible to the action configuration, that is passed to the `handle()` method.

```
<exception-handler
  class="de.odysseus.calyxo.control.misc.SimpleExceptionHandler">
  ...
  <param name="target" value="failure"/>
</exception-handler>
```

configures an exception handler to handle any type of exception.

Using a custom exception handler

The `ExceptionHandler` interface contains the methods `init(...)` and `handle(...)`. Assume, you implemented an exception handler in `org.foo.BarExceptionHandler`.

```
<exception-handler type="java.sql.SQLException"
  class="org.foo.BarExceptionHandler">
  <dispatch path="/WEB-INF/jsp/exception.jsp"/>
</exception-handler>
```

configures your exception handler to handle SQL exceptions.

3.1.6.6. Action Configurations

Actions are configured by `<action>` elements, which appear inside the `<actions>` element.

In short, an action configuration may specify a core action implementation, parameters, a sequence of filters, exception handlers and a dispatcher. Each action is associated with a module-relative path.

More concrete, an action configuration may take the following attributes:

1. The mandatory path giving the module-relative action path. The path must start with a slash (`'/'`). At most one action may state `path="/*"`, making it the default action. It will be invoked whenever a request did not match any other action path.

2. The optional `class` attribute gives the fully qualified class name of the core action command. If the `class` attribute is omitted, a built in default implementation will answer the dispatch configuration corresponding to the `<dispatch>` element whose name is referenced by the target attribute.
3. An optional `dispatcher` attribute to name a dispatcher to be used for this action. This may be overridden by `<dispatch>` elements.

The body of the `<action>` element contains a sequence of one or more

1. `<param>` elements to define named action configuration parameters
2. `<filter>` elements to add filters to the action's filter chain
3. `<dispatch>` elements to define dispatch targets
4. `<exception-handler>` elements to define local exception handlers

An action's configuration is reflected by the `ActionConfig` interface. An object implementing this interface is made available to action, filter and exception handler implementations. Beside methods corresponding one-to-one to an `<action>` element's attributes and children, the method `findDispatchConfig(String)` is used to lookup a dispatch configuration by name, consulting local dispatch configurations prior to global dispatch configurations.

Examples

In the Java code snippets in the following examples, `config` references an instance of `ActionConfig`.

Simple action

A simple action configuration might look like this:

```
<action path="/login" class="org.foo.bar.LoginAction">
  <dispatch name="success" action="/welcome"/>
  <dispatch name="failure" path="/WEB-INF/jsp/login.jsp"/>
</action>
```

Here, we associated path `/login` with an action. The action's core implementation is given by class `org.foo.bar.LoginAction`. The action contains two `<dispatch>` elements: one of these dispatch configurations may be chosen by the action implementation as a result. The first dispatch element says "dispatch to the action (in the same module) associated with path `/welcome`". The second dispatch element says "dispatch to the context-relative resource path `/WEB-INF/jsp/login.jsp`".

E.g., an action implementation may choose dispatch `success` using `config.findDispatchConfig("success")`.

Adding a Parameter

In the following example, we'll add a parameter named `users`.

```
<action path="/login" class="org.foo.bar.LoginAction">
  <param name="users" value="/WEB-INF/users.xml"/>
</action>
```

```
<dispatch name="success" action="/welcome"/>
<dispatch name="failure" path="/WEB-INF/jsp/login.jsp"/>
</action>
```

The action implementation may access this parameter using `config.getParamConfig("users")`.

Using the Default Action

If we omit the `class` attribute, the default action will be used.

```
<action path="/login.out">
  <dispatch path="/WEB-INF/jsp/login.jsp"/>
</action>
```

This action will just dispatch to `/WEB-INF/jsp/login.jsp`.

3.2. Reference

3.2.1. 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://calyx0.odysseus.de/xml/ns/control>

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

```
<!DOCTYPE calyx0-control-config
  PUBLIC "-//Odysseus Software GmbH//DTD Calyx0 Control 0.9//EN"
  "calyx0-control-config.dtd">
```

Elements

Name	Description
<code>calyx0-control-config</code>	Root element of a <i>Calyxo Control</i> configuration file.
<code>plugins</code>	Container element for plugin definitions.
<code>plugin</code>	Defines a controller plugin.
<code>dispatches</code>	Container element for global dispatch definitions.
<code>dispatch</code>	Defines a dispatch configuration.
<code>exception-handlers</code>	Container element for global exception handler definitions.
<code>exception-handler</code>	Defines an exception handler.

actions	Container element for action definitions.
action	Defines an action.
filter	Defines an action filter.
param	Defines a parameter.

3.2.1.1. The <calyxo-control-config> Element

Purpose

The <calyxo-control-config> element is the root element of a *Calyxo Control* 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 <code>http://calyxo.odysseus.de/xml/ns/control</code> .
version	NMTOKEN	Required - DTD/Schema version number. Must be 0.9.

Body

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

```
(base:import*, (base:functions | base:set | base:use)*,
 plugins?, dispatches?, exception-handlers?, actions?)
```

The first four elements are common to all Calyxo components. They are described in the [Calyxo Base configuration reference](#). The remaining elements and their children are described in the following sections.

Related elements

<plugins>, <dispatches>, <exception-handlers>, <actions>

3.2.1.2. The <plugins> Element

Purpose

The <plugins> element acts as a container for <plugin> elements.

Attributes

The <plugins> element has no attributes.

Body

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

(plugin*)

Related elements

<plugin>

3.2.1.3. The <plugin> Element

Purpose

The <plugin> element defines a module controller plugin.

Attributes

Name	Type	Description
<i>class</i>	CDATA	Required, Dynamic - The name of a class implementing the Plugin interface.

Body

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

(param*)

Related elements

<param>, <plugins>

3.2.1.4. The <dispatches> Element

Purpose

The <dispatches> element is a container for global <dispatch> elements. A global dispatch configuration is visible to all actions and may be overridden in actions by defining a local dispatch configuration with the same name.

Attributes

The <dispatches> element has no attributes.

Body

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

(dispatch*)

Related elements

<dispatch>

3.2.1.5. The <dispatch> Element**Purpose**

The <dispatch> element defines a dispatch configuration.

It may appear either inside the <dispatches> element to define a *global* dispatch configuration or inside an <action> element to define a *local* dispatch configuration.

A <dispatch> element requires either the path or action attribute to specify the dispatch target.

Attributes

Name	Type	Description
name	NMTOKEN	The dispatch name. The name attribute must be unique under <dispatch> siblings. Omitting the name sets the corresponding configuration property to null. Therefore, only one sibling may do this.
path	CDATA	Dynamic - The dispatch path. The interpretation of this attribute may depend on the dispatcher. The default dispatcher assumes a context-relative path, if it starts with a slash ('/'). Otherwise, if the redirection flag is set, it assumes an absolute URL. Otherwise, a path relative to the current request path is assumed.
action	CDATA	The dispatch action path. This is the path of the action, this configuration dispatches to. If the module attribute is set, it specifies the name of the module containing the action; otherwise, the current module is assumed. Either the path or action attribute must be set.
module	NMTOKEN	The dispatch action module. See above. The module attribute requires the action attribute to be set.
redirect	true false	Instruct the dispatcher to use redirection, if set to true. The default is false.
dispatcher	NMTOKEN	Specifies a custom dispatcher to be used.

Body

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

(param*)

Related elements

<param>, <dispatches>, <action>

3.2.1.6. The <exception-handlers> Element**Purpose**

The <exception-handlers> element is a container for global <exception-handler> elements. A global exception handler configuration is visible to all actions and may be overridden in actions by defining a local exception handler configuration for the same exception type.

Attributes

The <exception-handlers> element has no attributes.

Body

The body of the <exception-handlers> element is defined by the following sequence:

(exception-handler*)

Related elements

<exception-handler>

3.2.1.7. The <exception-handler> Element**Purpose**

The <exception-handler> element defines an exception handler.

It may appear either inside the <exception-handlers> element to define a *global* exception handler or inside an <action> element to define a *local* exception handler.

Attributes

Name	Type	Description
class	CDATA	Required, Dynamic - The name of a class implementing the Plugin interface.

type	CDATA	The name of the exception class, the exception handler should handle. Defaults to <code>java.lang.Exception</code> .
------	-------	--

Body

The body of the `<exception-handler>` element is defined by the following sequence:

`(param*)`

Related elements

`<param>`, `<exception-handlers>`, `<action>`

3.2.1.8. The `<actions>` Element**Purpose**

The `<actions>` element is a container for `<action>` elements.

Attributes

The `<actions>` element has no attributes.

Body

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

`(action*)`

Related elements

`<action>`

3.2.1.9. The `<action>` Element**Purpose**

The `<action>` element defines an action.

Attributes

Name	Type	Description
path	CDATA	Required - The module-relative action path, starting with a slash ('/'). One action may specify "/*" to be used as default action for requests, whose module-relative path does not match any other action path.
class	CDATA	Dynamic - The name of a class implementing the Action interface. If omitted, the module will use the default action class.
dispatcher	NMTOKEN	Specifies a custom dispatcher to be used. May be overridden by dispatch configurations.

Body

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

(param*, filter*, dispatch*, exception-handler*)

Related elements

<param>, <filter>, <dispatch>, <exception-handler>, <actions>

3.2.1.10. The <filter> Element**Purpose**

The <filter> element defines an action filter.

Either the class attribute or the name attribute can be used to specify the filter class.

The when attribute may contain an EL expression to conditionally include the filter during action processing.

Attributes

Name	Type	Description
class	CDATA	Dynamic - The name of a class implementing the Filter interface.
name	NMTOKEN	The name, under which a filter class has been registered by a plugin.
when	CDATA	Dynamic - An EL expression (without "\${" and "}"). If given, the expression is evaluated to a boolean value during action processing. If it evaluates to false, the module skips the filter when executing the filter chain.

Body

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

```
(param*,dispatch*)
```

Related elements

```
<param>, <dispatch>, <action>
```

3.2.1.11. The <param> Element**Purpose**

The <param> element defines a parameter by associating a name with a string value. <param> elements may appear inside <plugin>, <action>, <filter>, <dispatch> and <exception-handler> elements.

Attributes

Name	Type	Description
name	NMTOKEN	Required - The parameter name.
value	CDATA	Required, Dynamic - The parameter value.

Body

The <param> element has no body.

Related elements

```
<plugin>, <dispatch>, <exception-handler>, <action>, <filter>
```

3.2.2. Accessors

The `calyxo.control` accessors provides access to data related to the *Calyxo Control* 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.

3.2.2.1. The control.errors accessor

errors

Answers a `de.odysseus.calyxo.base.Messages` instance containing error messages by delegating to the module's message support.

Example

The expression `${calyxo.control.errors.allMessages}` evaluates to an iterator over the action error messages, that have been saved using the module's message support instance. Thus,

```
<ul>
  <c:forEach var="message" items="${calyxo.control.errors.allMessages}">
    <li>${calyxo.base.i18n.format[message]}</li>
  </c:forEach>
</ul>
```

renders all errors into an unordered list.

3.2.2.2. The control.warnings accessor

warnings

Answers a `de.odysseus.calyxo.base.Messages` instance containing warning messages by delegating to the module's message support.

Example

The expression `${calyxo.control.warnings.allMessages}` evaluates to an iterator over the action warning messages, that have been saved using the module's message support instance. Thus,

```
<ul>
  <c:forEach var="message" items="${calyxo.control.warnings.allMessages}">
    <li>${calyxo.base.i18n.format[message]}</li>
  </c:forEach>
</ul>
```

renders all warnings into an unordered list.

3.2.2.3. The control.infos accessor

infos

Answers a `de.odysseus.calyxo.base.Messages` instance containing info messages by delegating to the module's message support.

Example

The expression `${calyxo.control.infos.allMessages}` evaluates to an iterator over the action info messages, that have been saved using the module's message support instance. Thus,

```
<ul>
  <c:forEach var="message" items="${calyxo.control.infos.allMessages}">
    <li>${calyxo.base.i18n.format[message]}</li>
  </c:forEach>
</ul>
```

renders all infos into an unordered list.

3.3. Miscellaneous

3.3.1. Adding Groovy Support

Calyxo Control may be easily configured to add support for the [Groovy](#) scripting language (requires Groovy JSR-03 or later).

You only have to specify a different class loader to your module context. Just use `groovy.lang.GroovyClassLoader` as the module's class loader and tell it to load groovy classes from `/WEB-INF/groovy` by adding the following to your configuration:

```
<use value="${moduleContext}">
  <property name="classLoader">
    <object class="groovy.lang.GroovyClassLoader">
      <constructor>
        <arg value="${moduleContext.classLoader}"/>
      </constructor>
      <method name="addClasspath">
        <arg>
          <member value="${moduleContext.servletContext}">
            <method name="getRealPath">
              <arg value="/WEB-INF/groovy"/>
            </method>
          </member>
        </arg>
      </method>
    </object>
  </property>
</use>
```

```
</object>  
</property>  
</use>
```

Now the module will search for user defined classes in `/WEB-INF/groovy` before falling back to the default class loading mechanism.

This means that you now can either use Groovy or Java to implement any of your classes that will be loaded by this module!

Groovy Accessors, EL Functions, Plugins, Actions, Filters, ...

4. Calyxo Forms

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.

4.1. 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.

4.1.1. 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://calyx0.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 calyx0-forms-config
  PUBLIC "-//Odysseus Software GmbH//DTD Calyx0 Forms 0.9//EN"
  "calyx0-forms-config.dtd">
```

Copies of the DTD and XSD are located at `CALYX0_HOME/calyxo-forms/conf/share/calyxo-forms-config.*`.

Document structure

The root element is `<calyx0-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://calyx0.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:

```
<calyx0-forms-config version="0.9"
  xmlns="http://calyx0.odysseus.de/xml/ns/forms"
  xmlns:base="http://calyx0.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](#).

4.1.2. 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 `RegexMatcher`, 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 its 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>
```

4.1.3. 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.

4.1.4. 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 the 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.

4.1.5. 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 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.

4.1.6. 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.

4.1.7. 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

```

4.1.8. 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 committed, 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.

4.1.9. 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 configuration 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 scenarios 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&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.)

4.1.10. 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 be 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 know how to lookup labels, so it simply uses list model keys as labels. Therefore, subclasses are probably left with the implementation 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"/>
    </convert>
    <message>
      <arg name="field" value="choice"/>
    </message>
  </field>
</input>

```

```

    </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>

```

```
</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="modell23" value="${calyxo.base.module.attribute['modell23']}" />
<forms:select name="choice" listModel="${modell23}">
  <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="modell23" value="${calyxo.base.module.attribute['modell23']}" />
<forms:select name="choice" listModel="${modell23}" varGroupModel="group">
  <c:if test="${group.selectedCount == 0}">
    <forms:listoption>Choose...</forms:listoption>
  </c:if>
  <forms:listoptions sort="label" />
</forms:select>
```

4.1.11. l18n

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.

...

4.2. Reference

4.2.1. 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
<code>calyxo-forms-config</code>	Root element of a <i>Calyxo Forms</i> configuration file.
<code>validators</code>	Contains definitions of custom validators (i.e., matchers, checkers or converters) to be used in the <code><forms></code> section.
<code>matcher</code>	Defines a custom matcher to be used in the <code><forms></code> section.
<code>converter</code>	Defines a custom converter to be used in the <code><forms></code> section.
<code>checker</code>	Defines a custom checker to be used in the <code><forms></code> section.
<code>property</code>	Declares or sets a property value of a validator.
<code>message</code>	Specifies an error message.
<code>arg</code>	Sets an argument of an error message.
<code>forms</code>	Contains a set of form definitions (optionally for a specified locale).
<code>form</code>	Contains the validation rules concerning a single HTML input form.
<code>field</code>	Defines the sequence of validators (and an optional error message) applied to a single HTML input field.
<code>match</code>	Applies a matcher to an input field.
<code>convert</code>	Applies a converter to an input field.
<code>check</code>	Applies a checker to an input field.
<code>input</code>	...
<code>assert</code>	Defines an assertion rule concerning one or more input fields of a form.

4.2.1.1. 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
<code>xmlns</code>	CDATA	Required - XML namespace. Must be <code>http://calyxo.odysseus.de/xml/ns/forms</code> .
<code>version</code>	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>

4.2.1.2. 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>`**4.2.1.3. The `<matcher>` Element****Purpose**

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

Attributes

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

Body

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

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

Related elements`<converter>`, `<checker>`, `<property>`, `<message>`**4.2.1.4. The `<converter>` Element****Purpose**

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

Attributes

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

Body

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

(property*, message?)

Related elements

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

4.2.1.5. 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>

4.2.1.6. 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
<code>name</code>	NMTOKEN	Required - The name of the property.
<code>value</code>	CDATA	Dynamic - The value of the property. This attribute is required when overriding a validator property in a <code><match></code> , <code><convert></code> or <code><check></code> element.
<code>final</code>	true false	States, if a validator property is final and must not be overridden. The default is false. Allowed only inside a <code><matcher></code> , <code><converter></code> or <code><checker></code> element.

Body

The `<property>` element has no body.

Related elements

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

4.2.1.7. 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>

4.2.1.8. 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>

4.2.1.9. 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>

4.2.1.10. 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>

4.2.1.11. 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>

4.2.1.12. 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>

4.2.1.13. 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>

4.2.1.14. 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>

4.2.1.15. 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>

4.2.1.16. 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>

4.2.2. 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.

4.2.2.1. 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`.

4.2.2.2. 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`.

4.2.3. 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
<code>debug</code>	Renders an HTML table showing the current form validation result.

HTML Tags

Name	Description
<code>form</code>	Defines an HTML form.
<code>text</code>	Renders a text input field.
<code>password</code>	Renders a password input field.
<code>hidden</code>	Renders a hidden input field.
<code>textarea</code>	Renders a textarea input field.
<code>checkbox</code>	Renders a checkbox input field.
<code>radio</code>	Renders a radio button input field.
<code>group</code>	Container for checkbox or radio items.
<code>checkboxitem</code>	Renders a checkbox input field contained in a group.
<code>radioitem</code>	Renders a radio button input field contained in a group.
<code>select</code>	Renders a menu or selection list.
<code>option</code>	Renders a select option.
<code>listoption</code>	Renders a select option obtained from a list model.
<code>listoptions</code>	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
<code>%coreattrs</code>	<code>id</code> , <code>class</code> , <code>style</code> , <code>title</code>
<code>%i18n</code>	<code>lang</code> , <code>dir</code>
<code>%events</code>	<code>onclick</code> , <code>ondblclick</code> <code>onmousedown</code> , <code>onmouseup</code> , <code>onmouseover</code> , <code>onmousemove</code> , <code>onmouseout</code> <code>onkeydown</code> , <code>onkeyup</code> , <code>onkeypress</code>
<code>%input</code>	<code>accesskey</code> , <code>alt</code> , <code>disabled</code> , <code>readonly</code> , <code>tabindex</code>

	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").

4.2.3.1. 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.

4.2.3.2. 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 <code>assertClass</code> attribute value set.)
<code>create</code>	If given, must be true or false. The form tag uses the <code>FormsSupport.getFormData(...)</code> method to retrieve the form data instance for its action path. The boolean <code>create</code> parameter instructs the forms support to lazily create a new form data instance, if none is present for the specified path. (Default: false.)
<code>enctype</code>	Mapped to the HTML attribute of the same name.
<code>errorClass</code>	Appended to the HTML <code>class</code> 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.
<code>errorStyle</code>	Appended to the HTML <code>style</code> 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 <code>errorClass</code> attribute value set.)
<code>method</code>	Mapped to the HTML attribute of the same name.
<code>name</code>	Mapped to the HTML attribute of the same name.
<code>onreset</code>	Mapped to the HTML attribute of the same name.
<code>onsubmit</code>	Mapped to the HTML attribute of the same name.
<code>target</code>	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.).

4.2.3.3. 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>

4.2.3.4. 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>

4.2.3.5. 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>

4.2.3.6. 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>

4.2.3.7. 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>

4.2.3.8. 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>`**4.2.3.9. 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
<code>assertClass</code>	Appended to the HTML <code>class</code> attribute of the contained items, if an assertion concerning this input group failed. (Default: equally named attribute value of the embedding form.)
<code>assertStyle</code>	Appended to the HTML <code>style</code> attribute of the contained items, if an assertion concerning this input group failed. (Default: equally named attribute value of the embedding form.)
<code>errorClass</code>	Appended to the HTML <code>class</code> attribute of the contained items, if the validation of this input group failed. (Default: equally named attribute value of the embedding form.)
<code>errorStyle</code>	Appended to the HTML <code>style</code> attribute of the contained items, if the validation of this input group failed. (Default: equally named attribute value of the embedding form.)
<code>listModel</code>	The group's list model. If given, the <code>value</code> attributes of contained checkbox and radio items are verified 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>

4.2.3.10. 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>

4.2.3.11. 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 <code><group></code> tag.

Body

The `<radioitem>` tag has no body.

Related tags

`<radio>`, `<checkboxitem>`, `<checkbox>`

4.2.3.12. 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 corresponds 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>`

4.2.3.13. 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 <code><select></code> tag.

Body

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

Related tags

`<select>`, `<listoption>`, `<listoptions>`

4.2.3.14. 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 <code><select></code> tag. If no value attribute is specified, the option is rendered as a "default option": the value attribute of the rendered HTML <code><option></code> 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 <code><select></code> 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>`

4.2.3.15. 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 <code>String</code> or <code>String[]</code> .
sort	Determines the sorting parameter used to bring the rendered option entries into an appropriate order. Possible values are <ul style="list-style-type: none"> "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>`

4.2.4. 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.

4.2.4.1. 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	RegexMatcher	Tries to match the input with a regular expression. Returns the matched portion of the input, if successful. Property: <ul style="list-style-type: none"> pattern - The regular expression.
trim	TrimMatcher	Strips off leading and trailing white space. It can not fail and has no properties.

4.2.4.2. 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 <code>BigDecimal</code> . It has no properties.
bigInteger	BigIntegerConverter	Parses and formats objects of type <code>BigInteger</code> . Property: <ul style="list-style-type: none"> radix - The radix of the representation (from 2 to 36). (Default value is 10.)
boolean	BooleanConverter	Parses and formats objects of type <code>Boolean</code> . Properties: <ul style="list-style-type: none"> 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 <code>Byte</code> . Properties: <ul style="list-style-type: none"> 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 <code>Calendar</code> . It has no properties.
checkbox	BooleanConverter	Convenience converter to be used with checkbox inputs. Parses and formats objects of type <code>Boolean</code> . Empty inputs are parsed to false. Properties: <ul style="list-style-type: none"> 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: <ul style="list-style-type: none"> • 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: <ul style="list-style-type: none"> • 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: <ul style="list-style-type: none"> • 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: <ul style="list-style-type: none"> • 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: <ul style="list-style-type: none"> • 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: <ul style="list-style-type: none"> • 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: <ul style="list-style-type: none"> • 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.

4.2.4.3. 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: <ul style="list-style-type: none"> • 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: <ul style="list-style-type: none"> • expression - EL expression (without surrounding "\${" and "}").
interval	RangeChecker	Checks a numerical value to be in the range specified by its properties, which are: <ul style="list-style-type: none"> • 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

4.3. Extension Points

4.3.1. 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>

```

```
<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.

4.4. 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.

4.4.1. 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.

4.4.2. 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](#).

5. Calyxo Panels

Developing and maintaining dynamic views is challenging, since there are often aspects of both the content and the layout that are common across multiple views. When content and layout are intertwined, it is harder to maintain and extend the views. Reuse and modularity also suffer when common code is duplicated across views.

Calyxo Panels realize the *Composite View* design pattern to enable management of views in a flexible and maintainable way.

The *Calyxo Panels* component enables the definition of views in a container/component manner. Pages are dynamically composed from a tree of page fragments. Panel definitions are centralized in an XML configuration file.

Reusability

As a major plus, *Calyxo Panels* bring the benefits of object oriented programming into the view layer. Since your page fragments are reusable units, there's no need for code duplication.

We'll refer to page fragments as *templates*.

A *panel* defines how to bundle templates into a concrete view. Panel definitions themselves may use inheritance by extending other panel definitions, which is also of great use.

Flexibility

Beyond linking to templates, panel definitions may specify parameters and lists, which are made available to templates through custom tags (and *Calyxo* accessors).

Yet flexible, the *Calyxo Panels* component is impressively small. The configuration introduces just four elements (including the root element). The tag library contains three tags and there's one accessor. This condensed design helps to make the component very easy to understand and use.

Maintainability and Consistency

With *Calyxo Panels*, view maintenance is greatly simplified. Adding a new view requires just adding a panel definition and providing the template(s) containing content unique to that view.

Avoiding duplication of content by reuse leads to a high level of consistency. Keeping a consistent layout is made much easier.

Portability

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

5.1. Panels Concepts

Many aspects of a web application's presentation layer are common to all views. For example, pages may be composed of header, footer, menu and content areas. To keep maintainability and consistency, reuse of view components is an important issue. Thus, we need a mechanism to define our views as compositions of reusable page fragments.

Templates

To achieve this, we split our views into *templates*, which are combined into pages at runtime. A template may include other templates and may be included by other templates. In this model, an actual view may be seen as a template tree.

To be reusable, a template uses symbolic names to specify templates to be included. The actual template path has to be resolved according to some definition. Such a definition defines a template tree mentioned before.

Panels

In Calyxo terminology, these definitions are called *panels*.

- Panels may be nested to arbitrary depth. We'll call a panels's children *subpanels*. We distinguish between *oplevel* panels and *nested* panels.
- A panel has a name. Since this is the symbolic name used by a template to include a child template, it must be unique under its panel siblings.
- A panel definition may extend a toplevel panel definition. That is, panel definitions are reusable by themselves through inheritance.
- A panel may pass parameters to its associated template. These parameters are made available to the template through a Calyxo accessor.
- Toplevel panel definitions may be localized. That is, a panel definition can be associated with a locale. During panel lookup this is taken into account using *generalization*.
- A panel is dynamically associated with a template path. The associated template is resolved at instantiation time using inheritance and generalization. This selection process assigns concrete paths to the subpanel names used by templates to include their child templates.

Instantiation

Toplevel panels are visible to the controller. In a *Calyxo Panels* environment, you can use their names as resource paths, just as you do with static resources. Calyxo will dynamically lookup subpanels and compose and render the corresponding templates. Thus, you may see toplevel panels as *virtual views*.

We call the process of selecting a panel and rendering its associated template *instantiation*. When a toplevel panel is requested for instantiation,

1. a stack is created with the panel pushed onto it
2. the template associated with the panel is resolved
3. the request is dispatched to the associated template

During instantiation, the panel element on top of the maintained stack is called the *current* panel. The stack itself is kept in request scope.

When a template is executed, it includes templates associated with subpanels of the current panel. When a nested panel is requested for instantiation,

1. the requested subpanel is looked up in the current panel
2. the subpanel is pushed onto the stack (making it the current panel)
3. the template associated with the subpanel is resolved
4. the request is dispatched to the associated template
5. the subpanel is removed from the stack by performing a pop

This way, we traverse the tree of associated templates and render the results into the current response.

5.1.1. Configuration

As usual, the *Calyxo Panels* 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://calyx0.odysseus.de/xml/ns/panels`.

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 Panels* document type:

```
<!DOCTYPE calyx0-panels-config
  PUBLIC "-//Odysseus Software GmbH//DTD Calyx0 Panels 0.9//EN"
  "calyx0-panels-config.dtd">
```

Copies of the DTD and XSD are located at `CALYX0_HOME/calyxo-panels/conf/share/calyxo-panels-config.*`.

Note

Even if your parser supports XSD, you may want to include the DTD declaration in your

documents. A reason can be the use of an XML editor, which supports code assistance only for DTDs. Regardless of that, Calyxo will try to use XML Schema validation during parse.

Document structure

The root element is `<calyxo-panels-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.

Following these common elements, the root contains one or more `<panels>` elements. A `<panels>` 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 `<panels>` section, without specifying a locale. Thus, our configuration document looks like this:

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

  <panels>

    <!-- panel definitions go here -->

  </panels>

</calyxo-panels-config>
```

A `<panels>` element contains one or more [panel definitions](#).

5.1.2. Panels

Panels may be declared as children of a `<panels>` element using `<panel>` elements. Since panels may be nested to arbitrary depth, `<panel>` elements can contain `<panel>` child elements defining their subpanels.

A `<panel>` element requires the `name` attribute. The name must be unique under the `<panel>`'s siblings. For toplevel panels, the application uses this name like a resource path to instantiate it; for nested panels, the name is used by templates to include a template associated with a subpanel of the current panel. To reflect this, we use the convention to assign path-like names to toplevel panels and identifier-like names to subpanels.

The optional `template` attribute may be used to specify the associated template.

The optional `super` attribute can be used to specify a toplevel panel to extend. This is where inheritance comes into play. See the [inheritance](#) section for more on this.

Let's illustrate this by giving an example. Consider the following panel definitions:

```
<panel name="/base.page" template="/WEB-INF/jsp/layout/page.jsp">
  <panel name="header" template="/WEB-INF/jsp/layout/header.jsp"/>
  <panel name="menu" template="/WEB-INF/jsp/layout/menu.jsp"/>
  <panel name="content"/>
  <panel name="footer" template="/WEB-INF/jsp/layout/footer.jsp"/>
</panel>
```

```
<panel name="/foo.page" super="/base.page">
  <panel name="content" template="/WEB-INF/jsp/content/foo.jsp"/>
</panel>
```

- Following our convention, we assigned path-like names to toplevel panels and identifier-like names to nested panels.
- The `/base.page` panel defines the subpanels `header`, `menu`, `content` and `footer`. The `template` attributes denote the associated templates.
- The `/base.page`'s `content` panel does not specify a template, making `/base.page` *abstract*. That is, `/base.page` cannot be used as a dispatch target. Rather, it acts as a base definition, which may be extended by other panels.
- The `/foo.page` panel extends `/base.page` by specifying `super="/base.page"` and associates a template with its `content` subpanel, making it *concrete*. Thus, `/foo.page` can be used as a dispatch path by the controller.
- The `/WEB-INF/jsp/layout/page.jsp` template instantiates subpanels using the `<panel>` tag from the custom tag library provided by the *Calyxo Panels* component like this:

```
<jsp:root version="2.0"
  xmlns:panels="http://calyx.odyseus.de/jsp/panels"
  xmlns:jsp="http://java.sun.com/JSP/Page">
  ...
  <panels:panel name="header"/>
  ...
  <panels:panel name="menu"/>
  ...
  <panels:panel name="content"/>
  ...
  <panels:panel name="footer"/>
  ...
</jsp:root>
```

To your application, concrete toplevel panels are of particular interest, because they can be instantiated by the controller. Technically spoken, you may dispatch to toplevel panels using their (path-like) names, just as you do with any resources.

Beside subpanels, a `<panel>` element may contain elements defining [parameters](#) and [lists](#).

5.1.3. Parameters

You may define parameters within your panel definitions using the `<param>` element. The `<param>` element takes a mandatory name attribute and an optional value attribute. The name attribute values must be unique under the `<panel>`'s `<param>` children. The optional value attribute defines the parameter value.

Omitting the value attribute leaves the parameter value undefined. Accessing an undefined parameter value will cause an exception to be thrown. Undefined parameters should be overridden by derived panels or specified by the template during inclusion using a dynamic parameter. Use `value="{null}"` to specify value null.

The parameters of the current panel are accessible by templates using a *Calyxo* accessor.

Consider the following example:

```
<panel name="..." template="/foobar.jsp">
  ...
  <param name="foo" value="hi!"/>
  <param name="bar"/>
  ...
</panel>
```

- The first `<param>` element defines the parameter `foo` with value `"hi!"`
- The second `<param>` element defines the parameter `bar` without specifying a value.
- Now, the associated template `/foobar.jsp`, may access the current panel's parameters from a JSP EL expression using the `{calyxo.panels.param[...]}` accessor:

```
<jsp:root version="2.0" xmlns:jsp="http://java.sun.com/JSP/Page">
  ...
  <!-- access parameter via EL -->
  <someTag someattr="{calyxo.panels.param['foo']}">...</someTag>
  ...
</jsp:root>
```

Dynamic parameters

It is also possible to specify *dynamic* parameters during panel inclusion. A template uses the `<param>` tag inside a `<panel>` tag like this:

```
<jsp:root version="2.0"
  xmlns:panels="http://calyxo.odysseus.de/jsp/panels"
  xmlns:jsp="http://java.sun.com/JSP/Page">
  ...
  <panels:panel name="/button">
    <panels:param name="label" value="Save"/>
    <panels:param name="name" value="save"/>
  </panels:panel>
  ...
```

```
</jsp:root>
```

During instantiation of an included panel, dynamic parameters are added to its namespace, overriding static parameter definitions from the panel's configuration.

5.1.4. Lists

Up to now, templates must "know" subpanels and parameters by name to use them. This isn't a problem in most circumstances. However, a template might want to render a menu tree, where items are passed through by the panel definition; or, it might want to layout news items in a column, where the blocks of content are defined by a list of subpanels.

To satisfy these needs, `<panel>` elements may contain `<list>` elements as children. A `<list>` element requires a name attribute. As with subpanels and parameters, the name attribute values must be unique under the `<panel>`'s `<list>` children.

A list contains a sequence of `<item>` elements. An `<item>` element contains a mixed sequence of `<panel>`, `<param>` and - you guessed it - `<list>` elements.

A template uses the `<list>` tag to *iterate* over a list contained in the current panel. During iteration, the current item behaves as if its children elements were moved up to the current panel. That is - semantically - the current item is merged with the current panel.

As an example, consider the following panel definition:

```
<panel name="/blurb" template="/jsp/column.jsp">
  <list name="blurbs">
    <item>
      <param name="head" value="Item 1"/>
      <panel name="body" template="/jsp/blurb1.jsp"/>
    </item>
    <item>
      <param name="head" value="Item 2"/>
      <panel name="body" template="/jsp/blurb2.jsp"/>
    </item>
    <item>
      <param name="head" value="Item 3"/>
      <panel name="body" template="/jsp/blurb3.jsp"/>
    </item>
  </list>
</panel>
```

- The `/blurb` panel defines the `blurbs` list, containing three items, each containing a parameter named `head` and a panel named `body`.
- The layout template `/jsp/column.jsp` iterates over the list:

```
<jsp:root version="2.0"
  xmlns:panels="http://calyx.odyseus.de/jsp/panels"
  xmlns:jsp="http://java.sun.com/JSP/Page">
  <panels:list name="blurbs">
```

```

    <h4><panels:param name="head"/></h4>
    <p>
      <panels:panel name="body"/>
    </p>
  </panels:list>
</jsp:root>

```

Thus, when rendering `/blurb`, the above template creates paragraphs containing contents of `/jsp/blurb1.jsp` with heading "Item 1", `/jsp/blurb2.jsp` with heading "Item 2" and `/jsp/blurb3.jsp` with heading "Item 3".

The `<panels:list>` element also accepts an optional `varStatus` attribute. If set, the tag will export a *list status* object to that attribute in page scope. The list status object provides methods `boolean isFirst()`, `boolean isLast()` and `int getIndex()`, which may be used to retrieve information about the position of the current item within the list.

5.1.5. Inheritance

As mentioned before, a panel may explicitly *extend* a toplevel panel by specifying the `super` attribute. A *derived* panel extends its *base* panel in the following ways:

- An attribute that appears in the derived panel overrides an attribute in the base panel with the same name.
- A parameter that appears in the derived panel overrides a parameter in the base panel with the same name.
- A list that appears in the derived panel overrides a list in the base panel with the same name.
- Attributes, parameters, lists and subpanels, that are omitted in the derived panel, are inherited from the base panel.
- A derived panel may specify attributes, add lists, parameters and subpanels, that do not appear in the base definition.
- Subpanels of the derived panel implicitly extend their counterparts (subpanels with the same name) in the base panel; all rules apply to them as well.

During instantiation, these rules are to be applied when resolving attributes, subpanels, parameters and lists in the current panel.

Let us examine an example to make this clear:

```

<panel name="/base">
  <panel name="nested">
    <param name="param1"/>
    <param name="param2" value="p2"/>
    <panel name="nestedInested"/>
  </param>
</panel>

<panel name="/derived" super="/base" template="/WEB-INF/derived.jsp">
  <panel name="nested" template="/WEB-INF/nested1.jsp">
    <param name="param1" value="p1"/>
  </panel>
</panel>

```

```

    <param name="param2" value="override p2"/>
    <param name="param3" value="add p3"/>
  </param>
  <panel name="nested2" template="/WEB-INF/nested2.jsp"/>
</panel>

<panel name="/concrete" template="/WEB-INF/derived2.jsp">
  <panel name="foo" super="/derived">
    <panel name="nested">
      <panel name="nestedInested" template="/WEB-INF/bar.jsp"/>
      <param name="param3" value="override p3"/>
    </panel>
  </panel>
</panel>
</panel>

```

Don't run away! It looks harder than it is... Let's work out what happens here:

- Toplevel panel `/base` defines an abstract subpanel `nested`. It is abstract, because it does not specify a `template` attribute and also, because it has an abstract subpanel `nestedInested`. Furthermore, the `nested` subpanel contains the undefined parameter `param1`.
- Panel `/derived` extends `/base` and assigns a `template` value to subpanel `nested`, defines a value for `nested`'s `param1`, overrides the `param2` parameter and adds the `param3` parameter. Finally, it adds `nested2`, another nested panel. However, `/derived` is still abstract, because it inherits the abstract `nestedInested` panel.
- The `/concrete` panel definition takes a `foo` subpanel, which is derived from `/derived`. The `foo` panel overrides `param3`. Since it assigns a `template` attribute to `nestedInested`, `foo` is concrete and thus `/concrete` is concrete.

5.1.6. I18n

The *Calyxo Panels* component supports locale-dependent variants of panel definitions. This advanced feature may be used to define different view 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 panel definitions are collected into separate groups of `<panels>` elements. According to the locale properties, the `<panels>` element accepts attributes `language`, `country` and `variant` to specify a locale for the contained `<panel>` elements.

During instantiation, the process of searching attributes, subpanels, parameters and lists not only uses inheritance, but also generalization: if the attribute or element we're looking for is not available for the desired locale, the search proceeds using the generalized locale. In particular, generalization is used:

- when looking up a toplevel panel
- when searching for a base panel
- when searching for an associated template
- when looking up a subpanel, parameter or list

Finally, let's consider some examples:

```
<panels>
  <panel name="/address.page" template="/WEB-INF/address.jsp">
    <param name="foo" value="bar"/>
  </panel>
</panels>
```

```
<panels language="de">
  <panel name="/address.page" template="/WEB-INF/address_de.jsp"/>
</panels>
```

Depending on the desired locale, instantiation of `/address.page` will use the appropriate definition. Generalization will cause parameter `foo` to be found when using a german locale.

```
<panels>
  <panel name="/base.page" template="/WEB-INF/base.jsp">
    ...
  </panel>

  <panel name="/derived.page" super="/base.page">
    ...
  </panel>
</panels>
```

```
<panels language="de">
  <panel name="/base.page">
    ...
  </panel>
</panels>
```

Depending on the desired locale, instantiation of `/derived.page` will use the appropriate definition of `/base.page` to inherit from. Furthermore, generalization will cause `/WEB-INF/base.jsp` to be identified as associated template.

5.2. Reference

5.2.1. 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://calyx0.odysseus.de/xml/ns/panels>

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

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

Elements

Name	Description
<code>calyxo-panels-config</code>	Root element of a <i>Calyxo Panels</i> configuration file.
<code>panels</code>	Define a set of panel definitions, optionally specify a locale.
<code>panel</code>	Define a panel.
<code>param</code>	Define a parameter.
<code>list</code>	Define a list.
<code>item</code>	Define a list item.

5.2.1.1. The <calyxo-panels-config> Element

Purpose

The <calyxo-panels-config> element is the root element of a *Calyxo Panels* 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
<code>xmlns</code>	CDATA	Required - XML namespace. Must be <code>http://calyxo.odysseus.de/xml/ns/panels</code> .
<code>version</code>	NMTOKEN	Required - DTD/Schema version number. Must be 0.9.

Body

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

```
(base:import*, (base:functions | base:set | base:use)*, panels*)
```

The first four elements are common to all *Calyxo* components. They are described in the [Calyxo Base configuration reference](#). The <panels> element and its children are described in the following sections.

Related elements`<panels>`**5.2.1.2. The <panels> Element****Purpose**

The `<panels>` element defines a set of panel definitions.

Attributes

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

Body

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

`(panel*)`**Related elements**`<panel>`**5.2.1.3. The <panel> Element****Purpose**

The `<panel>` element defines a panel.

Attributes

Name	Type	Description
name	NMTOKEN	Required - The name of the panel.
super	NMTOKEN	Name of a toplevel panel, from which this panel inherits.

<code>template</code>	CDATA	Dynamic - The panel's template path.
-----------------------	-------	--------------------------------------

Body

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

`(param | panel | list)*`

Related elements

`<param>`, `<panel>`, `<list>`

5.2.1.4. The `<param>` Element**Purpose**

The `<param>` element defines a parameter.

Attributes

Name	Type	Description
<code>name</code>	NMTOKEN	Required - The parameter name.
<code>value</code>	CDATA	Dynamic - The parameter value.

Body

The `<param>` element has no body.

Related elements

`<panel>`, `<item>`

5.2.1.5. The `<list>` Element**Purpose**

The `<list>` element defines a list of items. Each item may contain panels, params and lists.

Attributes

Name	Type	Description
<code>name</code>	NMTOKEN	Required - The name of the list.

Body

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

```
(item*)
```

Related elements

```
<item>
```

5.2.1.6. The `<item>` Element

Purpose

The `<item>` element defines a list item.

Attributes

The `<item>` element has no attributes.

Body

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

```
(param | panel | list)*
```

Related elements

```
<param>, <panel>, <list>
```

5.2.2. Accessors

The `#{calyxo.panels}` accessors provides access to data related to the *Calyxo Panels* 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`.

5.2.2.1. The panels.param accessor

param[name]

Searches for a [parameter](#) with the specified name in the current panel. If the parameter exists, it evaluates to its value. Otherwise, an Exception is thrown.

Example

The expression `${calyxo.panels.param['foo']}` searches the current panel for a parameter named `foo` and evaluates to its value.

5.2.3. Tag Library

The *Calyxo Panels* custom tag library contains tags to include panels, access parameters and iterate over lists. In a JSP file, just associate the prefix you want to use for the tags with URI `http://calyxo.odysseus.de/jsp/panels`:

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

Since the tag library descriptor is contained in the *Calyxo Panels* 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-panels/conf/share/calyxo-panels.tld`.

Tags

Name	Description
<code>panel</code>	Include a subpanel (or toplevel panel).
<code>param</code>	Pass a dynamic parameter value to a panel.
<code>list</code>	Iterate over a list of items.

5.2.3.1. The <panel> Tag

Purpose

Include a subpanel of the current panel or a toplevel panel by name.

Attributes

Name	Description
name	Required - Specifies the name of a panel to be searched in the current panel.

Body

The <panel> tag may have a body containing <param> tags. The tag body is evaluated, but not included.

Related tags

<list>, <param>

5.2.3.2. The <param> Tag**Purpose**

Pass a dynamic parameter to a panel.

Requirements

A <param> tag has to be embedded in a <panel> tag.

Attributes

Name	Description
name	Required - Specifies the name of the parameter.
value	Required - Specifies the value of the parameter.

Body

The <param> tag has no body.

Related tags

<panel>, <list>

5.2.3.3. The <list> Tag**Purpose**

Iterate over a list contained in the current panel.

Attributes

Name	Description
name	Required - Specifies the name of a list to be searched in the current panel.
varStatus	If set, specifies the name of a page scope attribute. The tag will export a <i>list status</i> object to that attribute. The list status object has methods <code>boolean isFirst()</code> , <code>boolean isLast()</code> and <code>int getIndex()</code> , which may be used to retrieve information about the position of the current item.

Body

The `<list>` tag has no body.

Related tags

`<panel>`, `<param>`

5.3. Integration

In order to use the *Calyxo Panels* 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.

In either environment, the plugins modify the controller's dispatch behavior: before the controller dispatches a request to a specified path (through a forward or include), the plugin checks, if that path is the name of a toplevel panel definition. If so, it causes the controller to dispatch to the panel's associated template, instead. Otherwise, the request is dispatched to the specified path, as usual.

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

5.3.1. Panels Plugin for Calyxo Control

The plugin provides a *dispatcher* named panels for the *Calyxo Control* component. The panels dispatcher can be installed as the default dispatcher for the module. Alternatively, the panels dispatcher can be specified for a single action element or even for a single dispatch element.

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.panels.control.PanelsPlugin">
    <param name="config" value="/WEB-INF/calyxo-panels-config.xml"/>
    <param name="global" value="true"/>
  </plugin>
  ...
```

`</calyxo-control-config>`

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

If the optional `global` parameter is set to "true", the panels dispatcher is installed as the default dispatcher.

5.3.2. Panels Plugin for Struts

The panels plugin for Struts enables full use of the *Calyxo Panels* component with Struts. Plugin configuration details are covered in the [Calyxo Struts' Panels Plugin](#) documentation.

6. Calyxo Struts

The *Calyxo Struts* component focusses on integrating *Calyxo* components into the very popular Struts framework by the Apache Group.

Calyxo is for you, the Struts developer, if

- you like the core controller part of Struts, but feel strange about Struts Tiles, the Struts Validator and Struts' JSP taglibs; you're definitely right.
- you like all of Struts, including Tiles and the Validator; you'll love *Calyxo* anyway!
- your boss likes Struts; *Calyxo* will try to prevent you from suffering...

Calyxo Struts provides plugins to fully integrate the *Calyxo Base*, *Calyxo Panels* and *Calyxo Forms* components. *Calyxo Struts* requires Struts version 1.2.x or later. The current release has been tested with Struts versions 1.2.9 and 1.3.5.

Go ahead, Struts developer, have fun and take profit!

6.1. Calyxo Struts Plugins

Currently, the following plugins are available:

- the *Calyxo Base Struts plugin* to provide basic services like accessors,
- the *Calyxo Panels Struts plugin* as an alternative to Struts Tiles,
- the *Calyxo Forms Struts plugin* as an alternative to the Struts Validator.

6.1.1. Base plugin for Struts

The base plugin makes the *Calyxo Base* component available to Struts applications.

The plugin is a prerequisite to the other plugins provided by the *Calyxo Struts* component. It integrates *Calyxo* modules with Struts modules, *Calyxo*'s i18n support with Struts message resources and enables usage of *Calyxo* accessors.

Though, however, this plugin is mainly used in conjunction with other *Calyxo Struts* plugins, it may be used alone. This way, an application may load *Calyxo Base* configuration files and use all the accessors and JSP tags provided by *Calyxo Base*.

6.1.1.1. Struts Configuration

The plugin is loaded in the Struts configuration file:

```
<!DOCTYPE struts-config PUBLIC
  "-//Apache Software Foundation//DTD Struts Configuration 1.2//EN"
  "http://struts.apache.org/dtds/struts-config_1_2.dtd">

<struts-config>
```

...

```

<!-- load base plugin -->
<plug-in className="de.odysseus.calyxo.struts.base.BasePlugIn">
  <!-- optional configuration file -->
  <set-property property="config" value="/WEB-INF/calyxo-base-config.xml"/>
  <!-- optional module name -->
  <set-property
    property="module"
    value="main"/>
</plugin>

...

</struts-config>

```

The base plugin takes an optional config parameter to specify a *Calyxo Base* configuration file.

The optional module parameter can be used to specify the *Calyxo* module name. If omitted, the Struts module prefix is taken as module name.

The *Calyxo Base* plugin is a prerequisite to all the other plugins provided by the *Calyxo Struts* component. In the configuration, it must be located *before* all other *Calyxo Struts* plugins.

6.1.1.2. Struts modules

Struts modules are fully supported. The *Calyxo Struts* component provides the `StrutsModuleContext` module context implementation. This class supports URL prefix- as well as extension-mapped modules.

Here are the relevant details of the Struts module context implementation:

- `getName()` answers the module prefix, which is the empty string for the default module and a string starting with a slash ("/") for other modules.
- `getInitParameter(String name)` simply answers the servlet configuration's init parameter for the given name.
- `getPath(String action)` answers a context-relative path for the specified action mapping path. The action may have a query and/or anchor appended.
- Module scope is simulated by mapping key "<k>" to attribute "<k><p>" in application scope, where <p> is the module prefix. Thus, `module.getAttribute("foo")` is equivalent to `module.getServletContext().getAttribute("foo" + module.getName())`.

The module scope doesn't prevent conflicts to application scope. In particular, module scope and application scope are the same in the default module. The reason we implemented it this way is that Struts uses the same mapping strategy to save module-related stuff, e.g. message resources and configuration data.

6.1.1.3. Struts i18n

The `StrutsI18nSupport` class extends the *Calyxo Base* i18n support base class and will be used by *Calyxo* to resolve localized resources and to format messages.

The implementation simply forwards to the Struts message resources mechanism, so there's not much to say, here.

For example, an accessor expression like `${calyxo.base.i18n.bundle['foo'].resource['bar']}` will lookup key "bar" in the Struts message resource, that you registered in your Struts configuration with key="foo".

You may refer to the default message resource (which has no key attribute set in its configuration) by specifying bundle null.

6.1.2. Panels plugin for Struts

The panels plugin makes the *Calyxo Panels* component available to Struts applications. Please, consult the [Calyxo Panels documentation](#) to learn what you can do with panels.

6.1.2.1. Struts Configuration

The panels plugin requires the [base plugin](#) to be installed. In the Struts configuration, the base plugin must be declared *before* the forms plugin:

```
<!DOCTYPE struts-config PUBLIC
  "-//Apache Software Foundation//DTD Struts Configuration 1.2//EN"
  "http://struts.apache.org/dtds/struts-config_1_2.dtd">

<struts-config>

  ...

  <!-- load base plugin -->
  <plug-in className="de.odysseus.calyxo.struts.base.BasePlugIn"/>

  <!-- load panels plugin -->
  <plug-in className="de.odysseus.calyxo.struts.panels.PanelsPlugIn">
    <set-property property="config" value="/WEB-INF/calyxo-panels-config.xml"/>
  </plug-in>

</struts-config>
```

The panels plugin takes a mandatory config parameter to specify the *Calyxo Panels* configuration file.

Struts 1.3

Struts 1.3 requires to configure the request processing chain via XML in order to use *Calyxo Panels*. The *Calyxo Struts* jar contains a chain definition which emulates Struts 1.2 and adds

the Calyxo Panels commands. The chainConfig module parameter must be set to use this configuration instead of the default:

```
<init-param>
  <param-name>chainConfig</param-name>
  <param-value>
    de/odysseus/calyxo/struts/panels/calyxo-panels-chain-config.xml
  </param-value>
</init-param>
```

Note

The above example configures the default module's chain. If you're using Struts modules, module foo will take parameter foo/chainConfig, module bar will take parameter bar/chainConfig and so on.

6.1.2.2. Customizing the Request Processor

When the Struts controller dispatches a request to a specified path (through a forward or include), the plugin checks, if that path is the name of a toplevel panel definition. If so, it causes the controller to dispatch to the panel's associated template, instead. Otherwise, the request is dispatched to the specified path, as usual.

Struts 1.3

Struts 1.3 provides `org.apache.struts.chain.ComposableRequestProcessor` which allows to configure the request processing chain via XML. To use *Calyxo Panels* with this processor, the `de.odysseus.calyxo.struts.panels.PanelsCommand` must be added to the chain.

The *Calyxo Struts* jar contains a chain definition which emulates Struts 1.2 and adds the Calyxo Panels commands. Refer to the [configuration](#) section on how to use it. A copy of the modified chain config file is located in `calyxo-struts/conf/share/calyxo-panels-chain-config.xml`.

If you need to specify your own customized chain configuration, you will need to include the *Calyxo Panels* command yourself.

Struts 1.2

Under Struts 1.2.x, intercepting the process of request dispatching requires subclassing the module's request processor class. Therefore, the panels plugin comes with its own processor class. The plugin will automatically cause Struts to use this processor class as default.

However, if an application wants to use a custom processor class, it must extend the plugin's processor class!

In the Struts configuration file, the request processor class may be specified using the `processorClass` attribute of the `<controller>` element. For the panels plugin to work, the

request processor class used must be the same or a subclass of `de.odysseus.calyxo.struts.panels.PanelsRequestProcessor`.

6.1.3. Forms plugin for Struts

The forms plugin makes the *Calyxo Forms* component available to Struts applications. Please, consult the [Calyxo Forms documentation](#) to learn what forms can do for you.

Though the *Calyxo Forms* component comes with its own form/input JSP tag library, the plugin supports use of the tags provided by Struts.

- The *Calyxo Forms* tags are to be used in conjunction with the generic [Calyxo Action Forms](#), which free the application developer from implementing an action form class per HTML form.
- The Struts tags are to be used in conjunction with [Struts Legacy Forms](#). Plain action forms as well as DynaForms are supported.

6.1.3.1. Struts Configuration

Plugin

The forms plugin requires the [base plugin](#) to be installed. In the Struts configuration, the base plugin must be declared *before* the forms plugin:

```
<!DOCTYPE struts-config PUBLIC
  "-//Apache Software Foundation//DTD Struts Configuration 1.2//EN"
  "http://struts.apache.org/dtds/struts-config_1_2.dtd">

<struts-config>

  ...

  <!-- load base plugin -->
  <plug-in className="de.odysseus.calyxo.struts.base.BasePlugIn"/>

  <!-- load forms plugin -->
  <plug-in className="de.odysseus.calyxo.struts.forms.FormsPlugIn">
    <set-property property="config" value="/WEB-INF/calyxo-forms-config.xml"/>
  </plug-in>

</struts-config>
```

The forms plugin takes a mandatory config parameter to specify the *Calyxo Forms* configuration file.

Form Beans

Generally, a `<form-bean>`'s name attribute references a form in the *Calyxo Forms* configuration file.

Depending on if you want to use Struts' html tag library or the *Calyxo Forms* form/input tags, you will need to use different action form classes. Please refer to the sections on [Struts Legacy Forms](#) and [Calyxo Action Forms](#), respectively.

Actions

Struts `<action>` elements are configured for validation as usual.

6.1.3.2. Calyxo Action Forms

The *Calyxo Forms* component comes with its own tag library. This library contains custom tags related to the HTML form- and input tags. These tags directly access your form data, so form bean properties are ignored. To use these tags with Struts, you have to use the form bean class

```
de.odysseus.calyxo.struts.forms.CalyxoActionForm
```

or a subclass of it. The input values to be validated are directly taken from the request. Since *CalyxoActionForm* is a generic form class, Struts users should not derive from this class to add form bean properties as they are used to with ordinary action forms: the populated bean properties will neither be used as validation inputs, nor will the *Calyxo Forms* input tags pull their display values out of these properties.

Calyxo forms are declared in the Struts configuration file as usual:

```
<!DOCTYPE struts-config PUBLIC
  "-//Apache Software Foundation//DTD Struts Configuration 1.2//EN"
  "http://struts.apache.org/dtds/struts-config_1_2.dtd">

<struts-config>
  <form-beans>
    <form-bean
      name="loginForm"
      type="de.odysseus.calyxo.struts.forms.CalyxoActionForm"/>
    ...
  </form-beans>
  ...
</struts-config>
```

This also means, that you no longer need to implement dozens of form bean classes! The *CalyxoActionForm* class will keep the validated data in a map.

Warning

Again: this action form implementation does not use bean properties or mapped properties and does not work with the form input tags shipped with Struts! If you want to use action forms with the Struts tags, consult the [Struts "Legacy" Forms](#) section.

6.1.3.3. Struts "Legacy" Forms

Though [Calyxo Action Forms](#) are easier to use, there may be reasons to keep using the Struts HTML tags and form beans.

This might be the case if you're migrating from the Struts Validator to *Calyxo Forms*. It would be an extra effort to migrate your JSP pages to use *Calyxo Forms*' form/input tags. Or, you may depend on a special feature offered by some Struts tag, or...

Thus, we provide "legacy" action forms to be used with the Struts tag library, located in package `de.odysseus.calyxo.struts.forms.legacy`. There are two flavours of action form base classes:

- *Simple forms* – the classes `SimpleActionForm` and `SimpleDynaActionForm` are drop-in replacements for action form classes in existing applications.
- *Flushable forms* – the classes `FlushableActionForm` and `FlushableDynaActionForm` are intended for use in new projects, that want to stay with the Struts form/input tag library.

Note

Currently, there's one notable restriction: all form properties of legacy action forms that are to be validated by *Calyxo Forms* must be of type `java.lang.String`.

Simple Action Forms

Let's continue with the migration scenario. Consider you have an application which validates inputs by hand or uses the Struts Validator. Now, you would like to use *Calyxo Forms* to validate your forms. The migration path is short and simple 1-2-3 procedure:

1. Create your forms configuration file(s). Please, read the [Calyxo Forms documentation](#) to learn about that. The form names in the forms configuration correspond to form names in your Struts configuration.
2. Make your form beans subclasses of `SimpleActionForm`. Or, if you're using DynaForms, derive from `SimpleDynaActionForm`. You may also want to remove hand-coded validation code, that will now be covered by validation rules.
3. Load the *Calyxo Forms* for Struts Plugin as described in the [configuration](#) section.

You're done. Note, that you don't have to touch any action classes and JSP files!

The simple action forms have been designed for the "Enhance an existing application with minimal change effort" scenario. Thus, they do not need to offer things, that aren't available when using plain action forms or the Struts validator. In particular, committing form properties in a simple action form has no effect.

The major drawback of Struts action forms in general is, that they don't maintain a valid state. Thus, after an action form has once been populated with invalid data, this data will be shown again if the user navigates back to a page containing the form. Ugly!

Flushable Action Forms

The flushable action forms fix this. They are to be used with the Struts tags and keep validated form inputs for you.

Why are flushable forms called flushable? Flushable forms store committed form properties in a map. The *Calyxo Forms* component is able to format these values back into strings, that are to be displayed in the HTML form elements. However, the Struts tags pull their values out of the form bean properties, which still contain the unformatted input strings. So, when we *flush* our form, we format back our form data and re-populate the form bean properties. That way, we synchronize our view with the valid values. When redisplaying the form, these values will appear in a standard, localized format.

Flushing a form is triggered by the `<flush>` tag, provided by the *Calyxo Struts* component. The tag has to be placed inside a Struts form tag, *before* any input elements:

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0"
  xmlns:calyxo="http://calyxo.odysseus.de/jsp/struts"
  xmlns:html="http://struts.apache.org/tags-html">
  <html:form method="POST" action="/foo">
    <calyxo:flush/>
    Enter a number
    <html:text property="number"/>
    <html:submit/>
  </html:form>
</jsp:root>
```

6.1.3.4. Actions

Using *Calyxo Forms* with Struts doesn't require special action classes. However, you may want to interfere with *Calyxo Forms* when accessing validated form properties or committed form data. Everything starts by getting a reference to the module's `FormsSupport` instance:

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

Form Properties

The most essential thing is to access the validated form properties:

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

To commit the form properties, use

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

Please refer to the [Calyxo Forms Concepts](#) for further details.

Form Data

The FormsSupport instance may also be used 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());
```

or

```
support.removeFormData(request, "/checkin");
```

Please refer to the [Calyxo Forms Concepts](#) for further details.

6.1.3.5. Error Messages

Struts messages and *Calyxo* messages differ in that *Calyxo* messages may have a resource key *and* bundle name whereas Struts messages only have a key. Thus, when creating Struts action errors, the bundle "gets lost". To prevent from this, the validator plugin does the following:

If a Struts message has to be created from a *Calyxo* message with a bundle set, the plugin formats the message during validation and creates a Struts action message with key "", which takes the preformatted message string as an argument.

Therefore, to display validation error messages, you should either add the line

```
={0}
```

to your messages properties file, or check for messages with empty keys in your JSP and simply output their argument.

If you want to use Struts' <html:messages> tag to display your messages, you have to take the first option. Below is a JSP fragment which displays all messages using the struts.messages accessor provided by *Calyxo Struts*:

```
<c:set var="errors" scope="page"
  value="${calyxo.struts.messages['org.apache.struts.action.ERROR']}" />
<c:if test="${!empty pageScope['errors']}">
  <c:set var="bundle" value="${calyxo.base.i18n.bundle['application']}" />
  <ul>
    <c:forEach var="error" items="${errors}">
      <c:choose>
        <!-- formatted message (if you don't have line "{0}") -->
        <c:when test="${empty error.key}">
          <li>${error.values[0]}</li>
        </c:when>
        <!-- message without arguments -->
        <c:when test="${empty error.values}">
```

```
        <li>${bundle.message[error.key]}</li>
    </c:when>
    <!-- message with one or more arguments -->
    <c:otherwise>
        <li>${bundle.message[error.key][error.values]}</li>
    </c:otherwise>
</c:choose>
</c:forEach>
</ul>
</c:if>
```