Calyxo Intro

Table of contents

1. Welcome to Calyxo!

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).

2. General

2.1. 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.

2.2. 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 availabe 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.

2.3. 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`.

3. 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 peace 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.

3.1. 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.

3.1.1. 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.

3.1.2. 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.

3.1.3. 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"/>
  ```

3.1.4. 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.

3.2. 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.

3.2.1. 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.

3.2.2. 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.

3.3. 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.

3.3.1. 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://calyxo.odysseus.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>
```

3.3.2. 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 `calyxo.panels.param` accessor:

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

```
  ...
  <!-- access parameter via EL -->
  <sometag someattr="${calyxo.panels.param['foo']}">...</sometag>
  ...
</jsp:root>
```

3.3.3. 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?

3.3.4. 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://calyxo.odysseus.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"`.

3.3.5. 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.

3.4. 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 mutiple 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.

3.4.1. 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.

3.4.2. 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.

3.4.3. 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.

4. 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 devide 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/
```

4.1. 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.

4.1.1. 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.

4.1.2. 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?

4.1.3. 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.jspx</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!

4.2. 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.

4.2.1. Creating the JSP files

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

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.

4.2.2. Providing localized messages

Our login action produced error messages. Of cause, 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`.

4.3. 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.jspx</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.jspx`:

```
<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
```

4.4. 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...

4.5. 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 i18n 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.

4.5.1. 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 goodby page -->
    <action path="/goodbye">
      <dispatch path="/WEB-INF/jsp/goodbye.jspx"/>
    </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.jspx"/>
    </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.jspx"/>
    </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.*

4.5.2. 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.jspx</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.jspx`.

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.

4.6. 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.

4.6.1. 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.jspx">
      <!-- title (abstract) -->
      <param name="title"/>
      <!-- header -->
      <panel name="header" template="/WEB-INF/jsp/header.jspx">
        <param name="text" value="Login Sample App"/>
      </panel>
      <!-- menu -->
      <panel name="menu" template="/WEB-INF/jsp/menu.jspx">
        <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.jspx"/>
      <!-- footer -->
      <panel name="footer" template="/WEB-INF/jsp/footer.jspx"/>
    </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.jspx`.
- 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.jspx`.
- 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.jspx`.

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.jspx"/>
    </panel>

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

4.6.2. 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.jspx` 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.jspx"/>
      <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.jspx` 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.jspx` 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>
```

4.6.3. 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...

4.6.4. 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!

4.7. 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.jspx` 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.

4.7.1. 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="regexp">
          <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 `regexp` 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 `regexp` 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.

4.7.2. 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 failes. 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>
```

4.7.3. 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.jspx` as follows:

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

4.8. 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.

5. HOWTO's

5.1. 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.

5.2. 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*

5.3. 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
```