Calyxo Base

Table of contents

1. Introduction

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. 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. 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.

2.2. 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}`.

2.3. 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.

2.4. 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 varibale 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 oject given by expression ${list}, passing "bar" as an argument.

Please refer to the set, use & Co section for further details.

2.4.1. <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 invokation
- 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>

  <!-- 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>
```

3. Reference

3.1. Configuration

The elements described in the following sections are defined within namespace

```
http://calyxo.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 calyxo-base-config
  PUBLIC "-//Odysseus Software GmbH//DTD Calyxo Base 0.9//EN"
  "calyxo-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

```
<calyxo-foo-config version="0.9"
  xmlns="http://calyxo.odysseus.de/xml/ns/foo"
  xmlns:base="http://calyxo.odysseus.de/xml/ns/base">
  ...
</calyxo-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
`calyxo-base-config`	Root element of a standalone *Calyxo Base* configuration file.
`import`	Import another configuration file.
`functions`	Register a class defining static methods to be used as EL functions.
`set`	Define and store a variable.
`use`	Use an object to invoke methods and set properties.
`property`	Set a Java Bean property (or put an association into a `java.util.Map`).
`object`	Create and initialize a new object.
`member`	Access a method or field member.
`field`	Get a field value.
`method`	Invoke a method.
`arg`	Specify a method argument.

3.1.1. 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 `http://calyxo.odysseus.de/xml/ns/base`.
version	NMTOKEN	Required - DTD/Schema version number. Must be `0.9`.

Body

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

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

Related elements

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

3.1.2. 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 a an absolute URL.

Body

The <import> element has no body.

3.1.3. 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.

3.1.4. 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>

3.1.5. 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
value	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>

3.1.6. 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
name	NMTOKEN	Required - The name of the property (or association key).
value	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>

3.1.7. 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
class	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>

3.1.8. 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>

3.1.9. 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
class	CDATA	Dynamic - The fully qualified class name used to search for method or field members.
value	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>

3.1.10. 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
name	NMTOKEN	Required - The field name.

Body

The <field> element has no body.

(property*)

Related elements

<member>

3.1.11. 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>

3.1.12. 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>

3.2. 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.

3.2.1. 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>
  ...
  <img src="${calyxo.base.context.path}/images/logo.gif" alt="Logo"/>
  ...
</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()`.

3.2.2. 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"`.

3.2.3. 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 `I18nSuport.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, I18nSuport)`.

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']}
...
```

3.2.4. 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"`.

3.3. 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.

3.4. 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://calyxo.odysseus.de/jsp/base`:

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0"
  xmlns:base="http://calyxo.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 `CALYXO_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
`access`	Install the *Calyxo* accessors hierarchy in request scope.

HTML Tags

Name	Description
`a`	Render an HTML a tag that links to an action.
`form`	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
`%coreattrs`	`id, class, style, title`
`%i18n`	`lang, dir`

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

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

3.4.1. 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
`accesskey`	Mapped to the HTML attribute of the same name.
`action`	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 ?) and an anchor (starting with #).
`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 ommitted, 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.

3.4.2. 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.

3.4.3. 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.).

4. Extension Points

4.1. 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`.

4.2. 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 sceleton 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)}`.

5. 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.

6. Project

6.1. History of Changes

Version 0.9.0 (2006/10/28)

developer: **cbe** ***context:*** **docs** ***type:*** **update**
Minor documentation changes.

Version 0.9.0-rc3 (2006/02/12)

developer: **cbe** ***context:*** **code** ***type:*** **add**
Added `ModuleGroup` and `ModuleSelector` classes to allow both *Calyxo Struts* and *Calyxo Control* modules in one application.
developer: **cbe** ***context:*** **code** ***type:*** **add**
Added class `...util.ListOrderedMap`. Eliminated dependency to `commons-collections`.
developer: **cbe** ***context:*** **code** ***type:*** **update**
Accessor `base.module.forName[...]` now returns `null` if the given module does not exist (instead of throwing an exception).

Version 0.9.0-rc2 (2005/06/26)

developer: **cbe** ***context:*** **code** ***type:*** **add**
Added `ModuleContext.getClassLoader();` Added `ModuleContext.getExpressionEvaluator();` `Config` interface now extends `FunctionMapper`.
developer: **cbe** ***context:*** **code** ***type:*** **add**
Introduced `CalyxoException`.

Version 0.9.0-rc1 (2005/03/07)

developer: **cbe** ***context:*** **code** ***type:*** **update**
Renamed attribute `attribute` to `var` in <access> tag.
developer: **cbe** ***context:*** **code** ***type:*** **update**
Replaced `parseXyz(String)` by `toXyz(Object)` in `TypeFunctions`.
developer: **cbe** ***context:*** **code** ***type:*** **add**
Support for *Calyxo Base* configuration files.
developer: **cbe** ***context:*** **code** ***type:*** **add**
Redesigned variable/object related configuration elements. Removed `variable` and `store` elements. Added `set`, `use`, `object`, `member`, `field`, `method`, `constructor` and `arg` elements. Added `moduleContext`, `moduleScope` and `applicationScope` as implicit objects.
developer: **cbe** ***context:*** **code** ***type:*** **update**
New namespace `http://calyxo.odysseus.de/xml/ns/base`.

Version 0.9.0-b5 (2005/01/04)

developer: **cbe** ***context:*** **docs** ***type:*** **update**
Minor documentation changes.

Version 0.9.0-b4 (2004/11/21)

developer: **cbe** ***context:*** **code** ***type:*** **remove**
`DefaultI18nSupport` has been moved to `calyxo-control`.
developer: **cbe** ***context:*** **code** ***type:*** **fix**
Fix NPE in `I18nAccessor` when formatting messages without arguments.
developer: **cbe** ***context:*** **code** ***type:*** **fix**
Schema validation didn't work with `digester-1.6.jar`.

Version 0.9.0-b3 (2004/10/20)

developer: **cbe** ***context:*** **docs** ***type:*** **add**
Added documentation.
developer: **cbe** ***context:*** **code** ***type:*** **add**
Added `StandardFunctions` and `ModuleFunctions` EL function classes.
developer: **cbe** ***context:*** **code** ***type:*** **remove**
Removed message tag. Use the new base.i18n.format[...] accessor instead.
developer: **cbe** ***context:*** **code** ***type:*** **update**
Collected all base accessors under base.*
developer: **cbe** ***context:*** **code** ***type:*** **add**
Added base.eval[...] accessor class to evaluate dynamic EL expressions. Added CalyxoVariableResolver class.
developer: **cbe** ***context:*** **code** ***type:*** **add**
Added AccessException to be thrown by accessors.
developer: **cbe** ***context:*** **code** ***type:*** **update**
Now, variable and property configuration elements may take objects as values. For example, the expression `${module}` can be used to assign the module context to a bean property in a store element.

Version 0.9.0-b1 (2004/06/03)

developer: **cbe** ***context:*** **admin** ***type:*** **add**
First public release

6.2. Todo List

medium priority

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