Calyxo Control

Table of contents

1. Introduction

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

Calyxo Control versus Struts

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

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

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

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

2. Calyxo Control Concepts

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

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

Note

Please refer to the *Calyxo Base* Concepts section as a prerequisite to fully understand *Calyxo Control*.

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

Module Servlets

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

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

Modules

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

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

Actions

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

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

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

Filters

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

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

Exception Handlers

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

Dispatchers

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

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

Plugins

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

2.1. Modules

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

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

Module Context

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

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

Module Initialization

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

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

Once initialized, the module is ready for action processing.

Action Processing

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

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

2.1.1. Module Servlets

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

For each module, you have to

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

- a *prefix mapping* to map request URLs, whose context-relative paths start with some `"/prefix"`, to the module.

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

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

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

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

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

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

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

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

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

conflicts to occur.
Just consider the URL `http://localhost:8080/sample/foo/clash.bar`... which module servlet should be selected by the container?

2.2. Plugins

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

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

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

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

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

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

2.3. Actions

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

- a set of parameter configurations,
- a sequence of filters,
- a core action command,
- a set of dispatch configurations,
- a set of exception handlers,
- an optional dispatcher.

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

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

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

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

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

2.3.1. Action Commands

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

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

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

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

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

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

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

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

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

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

The controller instantiates exactly one instance per action configuration. As an important consequence, an action command can maintain its configuration as its internal state. On the

other hand, it has to be thread safe, since parallel invocations of the same action will execute the same action command instance.

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

Consider the following simple action configuration:

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

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

Directly implementing the Action interface

A direct `Action` implementation might look like this:

```
package org.foo.bar;

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

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

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

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

  /**
```

```
   * Execute action.
   */
  public DispatchConfig execute(
    HttpServletRequest request,
    HttpServletResponse response) throws Exception {

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

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

Inheriting from AbstractAction

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

```
package org.foo.bar;

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

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

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

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

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

```
    }
  }
}
```

Quite compact, isn't it?

2.3.2. Action Filters

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

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

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

During action invokation, the

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

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

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

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

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

Action filters may use the same techniques as servlet filters, including wrapping the request and/or the response.

A filter may decide to *not* execute the remaining chain. For example, if a filter doing user input validations detected invalid fields, it would want to answer some dispatch configuration to redisplay the last page, rather than continuing with command execution.

Finally, an action filter may inspect (and even replace) the result of the command execution.

2.4. Exception Handlers

Exception handlers implement the `de.odysseus.calyxo.control.Filter` interface. The controller instantiates one instance per exception handler configuration. After instantiation, the controller gives an exception handler the opportunity to do some initialization by calling its

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

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

Exception handlers may be declared for a hierarchy of exception classes either local to an action or global to the module. See the section on exception handler configuration for details.

If an action's invocation throws an exception, the module searches for an exception handler as follows: for each class *cls* on the inheritance path from the exception's class up to `java.lang.Exception`, the module first checks for a local, then for or global exception handler, that has been declared for the current class *cls*. On the first one it finds, it calls the

```
public DispatchConfig handle(
  HttpServletRequest request,
  HttpServletResponse response,
  ActionConfig actionConfig,
  Exception exception) throws IOException, ServletException;
```

method. The `actionConfig` parameter corresponds to the action, that threw the exception. The handler is responsible to answer a dispatch configuration.

If the module didn't find a matching exception handler it wraps the exception into a `ServletException` and throws it again. Similar, an exception, thrown in an exception handler's `handler()` method is *not* caught by the module and will go up to the servlet container.

2.5. Dispatchers

Dispatchers implement the `de.odysseus.calyxo.control.Dispatcher` interface, which defines the single method

```
public void dispatch(
  HttpServletRequest request,
  HttpServletResponse response,
  DispatchConfig config) throws IOException, ServletException;
```

The `de.odysseus.calyxo.control.conf.DispatchConfig` interface reflects a dispatch configuration and provides methods to access the dispatch path, module and action, the redirection flag as well as the dispatch parameters.

A custom dispatcher has to be registered by a plugin using either method of `de.odysseus.calyxo.control.PluginContext`:

- `setDispatcher(String, Dispatcher)` - to register a dispatcher under some name
- `setDefaultDispatcher(Dispatcher)` - to set the default dispatcher.

Custom dispatchers may be referenced by name in `<action>` and `<dispatch>` configurations through their `dispatcher` attribute.

An action invocation results in a dispatch configuration. The module selects a dispatcher as follows: if the dispatch configuration has a dispatcher set, use it; otherwise, if the action configuration has a dispatcher set, use it; otherwise, use the default dispatcher. In other words, the dispatch's dispatcher is most significant, followed by the action's dispatcher, followed by the default dispatcher.

If not overridden, *Calyco Control* uses `de.odysseus.calyxo.control.impl.DefaultDispatcher` as the default dispatcher. This dispatcher processes a dispatch configuration as follows:

1. if the dispatch configuration specifies an action (and a module),
   path := context-relative path of that action (in that module)
2. if the dispatch configuration specifies a path,
   path := that path
3. append the dispatch parameters to *path* as HTTP query parameters
4. if the redirection flag is set and *path* starts with `'/'`,
   prepend the context path to *path*
5. dispach using redirect, forward or include:
   - if the redirection flag is set, redirect the response to *path*
   - otherwise, if the response is not yet committed, forward to *path*
   - otherwise, include *path*

Custom dispatchers are free to reuse the default dispatcher by subclassing it or delegating to it.

2.6. Configuration

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

Namespace

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

Schema/DTD Validation

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

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

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

Document structure

The root element is `<calyxo-control-config>`. As common to all of *Calyxo*'s configuration files, the root element uses the `xmlns` attribute to specify the namespace and the `version` attribute to specify the schema/DTD version.

The root may optionally contain `<base:import>` elements, followed by `<base:functions>`, `<base:set>` and `<base:use>` elements. See the *Calyxo Base* configuration section for a description of these elements. Using these elements requires the declaration of namespace `http://calyxo.odysseus.de/xml/ns/base` for prefix `base` (as in the document template below).

Following these common elements, the root optionally contains

1. a `<plugins>` element containing `<plugin>` elements,
2. a `<dispatches>` element, containing global `<dispatch>` elements,
3. an `<exception-handlers>` element, containing global `<exception-handler>` elements,
4. an `<actions>` element, containing `<action>` elements.

Thus, our configuration document has a general structure like this:

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

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

  <plugins>
    <!-- plugin definitions go here -->
  </plugins>

  <dispatches>
    <!-- dispatch definitions go here -->
  </dispatches>

  <exception-handlers>
    <!-- exception handler definitions go here -->
  </exception-handlers>

  <actions>
    <!-- action definitions go here -->
```

```
  </actions>

</calyxo-control-config>
```

Each of the nested elements is optional and thus may be ommitted.

Configuration Interfaces

All configuration elements have corresponding Java interfaces. When actions, filters or exception handlers are initialized by the controller, an instance reflecting their corresponding configuration element is passed to them.

The configuration interfaces are located in package `de.odysseus.calyxo.control.conf`.

2.6.1. Parameter Configurations

Each of the `<plugin>`, `<action>`, `<filter>`, `<dispatch>` and `<exception-handler>` elements may be configured to take a set of parameters by specifying nested `<param>` elements.

Consequently, the `PluginConfig`, `ActionConfig`, `FilterConfig`, `DispatchConfig` and `ExceptionConfig` interfaces extend the `ParamsConfig` interface, which provides the `getParamConfig(String)` method to lookup a parameter configuration by its name.

A `<param>` element itself requires the two attributes

- `name`, specifying the parameter name, and
- `value`, specifying the parameter value.

The `value` attribute is dynamic (that is, its value may contain expressions) and of type string.

A parameter configuration is reflected by the `ParamConfig` interface, which defines the `getName()` and `getValue()` methods.

Example

The following action element defines parameter `users`.

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

Now, the `org.foo.bar.LoginAction` action implementation might access the corresponding `ParamConfig` like this:

```
import de.odysseus.calyxo.control.conf.ParamConfig;
import de.odysseus.calyxo.base.conf.ConfigException;

public class LoginAction extends de.odysseus.calyxo.control.misc.AbstractAction {
```

```
  private String users;

  public void init() throws ConfigException {
    ParamConfig param = getActionConfig().getParamConfig("users");
    if (param == null) {
      throw new ConfigException("Missing parameter 'users'!");
    }
    users = param.getValue();
  }
  ...
}
```

2.6.2. Plugin Configurations

A plugin is defined using the `<plugin>` element, which may appear inside the `<plugins>` element. It takes a mandatory `class` attribute, giving the name of a class implementing the `Plugin` interface.

The `<plugin>` element may contain any number of `<param>` elements.

A plugin configuration is reflected by the `PluginConfig` interface, which defines the `getParamConfig()` method to lookup parameters by name.

Example

Here's a plugin configuration for plugin class `org.foo.bar.FooPlugin`, passing parameter `bar`.

```
<plugin class="org.foo.bar.FooPlugin">
  <param name="bar" value="foobar"/>
</action>
```

The controller will instantiate an instance of `FooPlugin` during module load and call its

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

method. The `PluginContext` gains access to the module context and constains methods to register dispatchers and filter classes by name. See the general plugins section for details.

2.6.3. Filter Configurations

Filters are added to an action using the `<filter>` element. They appear as children of an `<action>` element, following the action's `<param>` elements.

The `class` attribute may be used to identify a class implementing the `Filter` interface. As an alternative, the `name` attribute may be used to reference a filter class, that has been registered by a plugin under some unique name.

The `when` attribute can be used to specify a JSP EL expression (without `"${"` and `"}"`) to conditionally execute the filter. During action invocation, the expression will be evaluated to a boolean value. If it results to `true`, the filter is executed as usual. Otherwise, the filter will be skipped and is not part of the filter chain for that request.

The `<filter>` element may contain any number of `<param>` elements as well as `<dispatch>` elements.

The `Filter` interface defines the `init()` and `filter()` methods. For an action, the controller instantiates the filter instances and arranges them into a sequence in the same order the `<filter>` elements appear within the `<action>` element.

Since a filter class may be registered by a plugin, it is not uncommon for filters to implement both, the `Plugin` and `Filter` interfaces.

Examples

Calyxo Control comes with some ready-to-use filters, that we'll use here to illustrate filter definitions.

Branch Filter

The *branch* simply dispatches to a target according to its configuration. It is used with a filter `when` condition. The filter may be attached to an action like this:

```
<action path="/login" class="org.foo.bar.LoginAction">
  <filter class="de.odysseus.calyxo.control.misc.BranchFilter"
    when="!empty param.forgotten">
    <dispatch action="/recoverPassword"/>
  </filter>
  ...
</action>
```

The filter also implements the `Plugin` interface to associate the filter with some name. The default filter name is `"branch"`. Therefore, to reference the filter by its default name, we add the plugin configuration

```
<plugin class="de.odysseus.calyxo.control.misc.BranchFilter"/>
```

to the `<plugins>` element. Now, we can rewrite the above action definition as

```
<action path="/login" class="org.foo.bar.LoginAction">
  <filter name="branch" when="!empty param.forgotten">
    <dispatch action="/recoverPassword"/>
  </filter>
  ...
</action>
```

The dispatch configuration may be either specified directly by a nested anonymous dispatch

configuration (as above) or by the `target` parameter referencing a dispatch configuration visible to the enclosing action:

```
<action path="/login" class="org.foo.bar.LoginAction">
  <filter name="branch" when="!empty param.forgotten">
    <param name="target" value="recover"/>
  </filter>
  ...
  <dispatch name="recover" action="/recoverPassword"/>
</action>
```

No-Cache Filter

The *no-cache* filter adds some HTTP response headers to prevent browsers from caching the page. The filter may be attached to an action like this:

```
<action path="/login" class="org.foo.bar.LoginAction">
  <filter class="de.odysseus.calyxo.control.misc.NoCacheFilter"/>
  ...
</action>
```

The filter also implements the `Plugin` interface to associate the filter with some name. The default filter name is `"no-cache"`. Therefore, to reference the filter by its default name, we add the plugin configuration

```
<plugin class="de.odysseus.calyxo.control.misc.NoCacheFilter"/>
```

to the `<plugins>` element. Now, we can rewrite the above action definition as

```
<action path="/login" class="org.foo.bar.LoginAction">
  <filter name="no-cache"/>
  ...
</action>
```

Cancel Filter

Consider a form containing a "Cancel" button. When the user presses this button, we usually do not want the core action implementation to handle this. The *cancel* filter checks for the existence of a request parameter, `"cancel"` by default, and - if it exists - directly dispatches to some target, without executing the remaining chain. Again, the default dispatch target name is `"cancel"`.

This could be done with the branch filter. However, since this is a common task, the filter is provided for convenience.

```
<action path="/register" class="org.foo.bar.RegisterAction">
  <filter class="de.odysseus.calyxo.control.misc.CancelFilter"/>
  ...
  <dispatch name="cancel" action="/goodbye"/>
</action>
```

Above, we used the filter's default "cancel" for the request parameter name and target dispatch name. If the request parameter name were "abort" and we wanted to use a global dispatch configuration named "goodbye", our action might look like this:

```
<action path="/register" class="org.foo.bar.RegisterAction">
  <filter class="de.odysseus.calyxo.control.misc.CancelFilter">
    <param name="parameter" value="abort"/>
    <param name="target" value="goodbye"/>
  </filter>
  ...
</action>
```

The CancelFilter class also implements the Plugin interface to associate the filter with some name. The default filter name is - no wonder - "cancel". Thus, after adding CancelFilter as a plugin, we could rewrite the above examples by replacing class="..." with name="cancel".

Finally, the target dispatch configuration may also be specified directly using a nested anonymous dispatch configuration element:

```
<action path="/register" class="org.foo.bar.RegisterAction">
  <filter name="cancel">
    <dispatch action="/goodbye"/>
  </filter>
  ...
</action>
```

2.6.4. Dispatch Configurations

The <dispatch> element is used to define a dispatch configuration. It may appear either inside the <dispatches> element to define a *global* dispatch configuration or inside an <action> element to define a *local* dispatch configuration.

A dispatch configuration contains information about how to proceed after action invocation. It may either point to another action or to a resource path. It may also specify a custom dispatcher and whether to use redirection.

The <dispatch> element takes the following attributes:

- The name attribute - specifying the name of the dispatch configuration. Actions use this name to lookup a configuration in the pool of local and global dispatch configurations. Omitting the name attribute creates a "default" dispatch configuration whose name property is null.
- The action attribute, optionally in conjunction with the module attribute - to dispatch to another action; if the module attribute is omitted, the current module is assumed.
- The path attribute - to dispatch to that path; e.g., the path may be a context-relative. However, custom dispatchers may interpret this attribute in their own fashion.
- The redirect attribute - specifying to redirect the response, if set to true.
- The dispatcher attribute - specifying a custom dispatcher to be used; custom dispatchers

are registered by plugins.

It is an error to specifying both, the `action` *and* `path` attributes. Furthermore, the `module` attribute is only permitted in conjunction with the `action` attribute.

The body of the `<action>` element may contain a set of `<param>` elements. Dispatchers may use parameters in their own way. However, the default dispatcher will append them as request parameters.

A dispatch configuration is reflected by the `DispatchConfig` interface, which defines methods to access the attribute values as well as the `getParamConfig()` method to lookup parameters by name.

The `DispatchConfig` interface is the result type of the action's `execute()` method. An action uses its action configuration, `ActionConfig`, to lookup a dispatch configuration by name, using the

```
public DispatchConfig findDispatchConfig(String name);
```

method. The default dispatch configuration can be accessed by specifying name `null`.

Examples

Dispatching to an action in the current module

```
<dispatch name="foo" action="/bar"/>
```

dispatches to action with `path="/bar"` in the current module.

Dispatching to an action in another module

```
<dispatch name="foo" module="other" action="/bar"/>
```

dispatches to action with `path="/bar"` in the module `other`.

Dispatching to a context-relative path

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

dispatches to JSP at `/WEB-INF/jsp/bar.jsp`.

Adding parameters

```
<dispatch name="foo" path="/WEB-INF/jsp/bar.jsp">
  <param name="mode" value="lazy"/>
</dispatch>
```

dispatches to JSP using request path `/WEB-INF/jsp/bar.jsp?mode=lazy`.

Using redirection

```
<dispatch name="foo" path="/jsp/bar.jsp" redirect="true"/>
```

redirects to JSP at context-relative path `/jsp/bar.jsp`, whereas

```
<dispatch name="foo" path="http://calyxo.org" redirect="true"/>
```

redirects to absolute URL `http://calyxo.org`.

Using a custom dispatcher

Consider you implemented a custom dispatcher, which wraps the response to capture and buffer its content, before dispatching to a resource producing some XML. Then, it gets the the xml content and and applies an XSL transformation on it. Say, you registered this accessor as `"xslt"` using a plugin. After all,

```
<dispatch name="foo" dispatcher="xslt" path="/bar.xml">
  <param name="stylesheet" value="/WEB-INF/xsl/style.xsl"/>
</dispatch/>
```

will cause your custom dispatcher to apply XSLT stylesheet `/WEB-INF/xsl/style.xsl` to the content of `/bar.xml`.

2.6.5. Exception Configurations

An exception handler is defined using the `<exception-handler>` element. It may appear either inside the `<exception-handlers>` element to define a *global* exception handler or inside an `<action>` element to define a *local* exception handler.

The `<exception-handler>` element takes a mandatory `class` attribute, giving the name of the exception handler class, which has to implement the `ExceptionHandler` interface. The optional `type` attribute specifies the name of the exception type, that will be treated by the handler. If omitted, a `java.lang.Exception` will be assumed.

The `<exception-handler>` element may contain any number of `<param>` and `<dispatch>` elements.

An exception configuration is reflected by the `ExceptionHandlerConfig` interface, which defines the `getParamConfig()` method to lookup parameters by name.

Exception handlers are instantiated and initialized when the module gets loaded, one for each configuration. If an action invocation throws an exception, the module catches it and searches for an exception handler configuration as follows: for each class *cls* on the inheritance path from the exception's class up to `java.lang.Exception`, it first checks for a local, then for or global exception configuration whose `type` attribute matches the current class *cls*. If it finds one, it gets the corresponding `ExceptionHandler` instance and invokes its `handle()` method. Otherwise, the exception is wrapped into a `ServletException` and thrown again.

Examples

The following examples show some exception handler configurations. Whether a handler is selected for a concrete exception, is determined by the module's selection algorithm as explained above and also depends on whether it is defined as a local or global exception handler.

Using the simple exception handler

Calyxo Control supplies a simple exception handler implementation. This generic exception handler requires the parameters `bundle` and `key` to identify a message template. It will create and save a message object into the current request, passing the exception message as an argument. Finally, it looks up and returns a dispatch configuration in the action identified by the `target` parameter.

```
<exception-handler
  class="de.odysseus.calyxo.control.misc.SimpleExceptionHandler">
  <param name="bundle" value="messages"/>
  <param name="key" value="exception"/>
  <dispatch path="/WEB-INF/jsp/exception.jsp"/>
</exception-handler>
```

The dispatch configuration may be either specified directly by a nested anonymous dispatch configuration (as above) or by the `target` parameter referencing a dispatch configuration visible to the action configuration, that is passed to the `handle()` method.

```
<exception-handler
  class="de.odysseus.calyxo.control.misc.SimpleExceptionHandler">
  ...
  <param name="target" value="failure"/>
</exception-handler>
```

configures an exception handler to handle any type of exception.

Using a custom exception handler

The `ExceptionHandler` interface contains the methods `init(...)` and `handle(...)`. Assume, you implemented an exception handler in `org.foo.BarExceptionHandler`.

```
<exception-handler type="java.sql.SQLException"
  class="org.foo.BarExceptionHandler">
  <dispatch path="/WEB-INF/jsp/exception.jsp"/>
</exception-handler>
```

configures your exception handler to handle SQL exceptions.

2.6.6. Action Configurations

Actions are configured by `<action>` elements, which appear inside the `<actions>` element.

In short, an action configuration may specify a core action implementation, parameters, a sequence of filters, exception handlers and a dispatcher. Each action is associated with a module-relative path.

More concrete, an action configuration may take the following attributes:

1. The mandatory `path` giving the module-relative action path. The path must start with a slash (`'/'`). At most one action may state `path="/*"`, making it the default action. It will be invoked whenever a request did not match any other action path.
2. The optional `class` attribute gives the fully qualified class name of the core action command. If the `class` attribute is omitted, a built in default implementation will answer the dispatch configuration corresponding to the `<dispatch>` element whose name is referenced by the `target` attribute.
3. An optional `dispatcher` attribute to name a dispatcher to be used for this action. This may be overridden by `<dispatch>` elements.

The body of the `<action>` element contains a sequence of one or more

1. `<param>` elements to define named action configuration parameters
2. `<filter>` elements to add filters to the action's filter chain
3. `<dispatch>` elements to define dispatch targets
4. `<exception-handler>` elements to define local exception handlers

An action's configuration is reflected by the `ActionConfig` interface. An object implementing this interface is made available to action, filter and exception handler implementations. Beside methods corresponding one-to-one to an `<action>` element's attributes and children, the method `findDispatchConfig(String)` is used to lookup a dispatch configuration by name, consulting local dispatch configurations prior to global dispatch configurations.

Examples

In the Java code snippets in the following examples, `config` references an instance of `ActionConfig`.

Simple action

A simple action configuration might look like this:

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

Here, we associated path `/login` with an action. The action's core implementation is given by class `org.foo.bar.LoginAction`. The action contains two `<dispatch>` elements: one of these dispatch configurations may be chosen by the action implementation as a result. The first dispatch element says "dispatch to the action (in the same module) associated with path

`/welcome`". The second dispatch element says "dispatch to the context-relative resource path `/WEB-INF/jsp/login.jsp`".

E.g., an action implementation may choose dispatch `success` using `config.findDispatchConfig("success")`.

Adding a Parameter

In the following example, we'll add a parameter named `users`.

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

The action implementation may access this parameter using `config.getParamConfig("users")`.

Using the Default Action

If we omit the `class` attribute, the default action will be used.

```
<action path="/login.out">
  <dispatch path="/WEB-INF/jsp/login.jsp"/>
</action>
```

This action will just dispatch to `/WEB-INF/jsp/login.jsp`.

3. Reference

3.1. Configuration

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

The elements described in the following sections are defined within namespace

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

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

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

Elements

Name	Description
calyxo-control-config	Root element of a *Calyxo Control* configuration file.
plugins	Container element for plugin definitions.
plugin	Defines a controller plugin.
dispatches	Container element for global dispatch definitions.
dispatch	Defines a dispatch configuration.
exception-handlers	Container element for global exception handler definitions.
exception-handler	Defines an exception handler.
actions	Container element for action definitions.
action	Defines an action.
filter	Defines an action filter.
param	Defines a parameter.

3.1.1. The <calyxo-control-config> Element

Purpose

The `<calyxo-control-config>` element is the root element of a *Calyxo Control* configuration file.

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

Attributes

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

Body

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

```
(base:import*, (base:functions | base:set | base:use)*,
 plugins?, dispatches?, exception-handlers?, actions?)
```

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

Related elements

<plugins>, <dispatches>, <exception-handlers>, <actions>

3.1.2. The <plugins> Element

Purpose

The <plugins> element acts as a container for <plugin> elements.

Attributes

The <plugins> element has no attributes.

Body

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

```
(plugin*)
```

Related elements

<plugin>

3.1.3. The <plugin> Element

Purpose

The <plugin> element defines a module controller plugin.

Attributes

Name	Type	Description
class	CDATA	Required, Dynamic - The name of a class implementing the Plugin interface.

Body

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

```
(param*)
```

Related elements

<param>, <plugins>

3.1.4. The <dispatches> Element

Purpose

The <dispatches> element is a container for global <dispatch> elements. A global dispatch configuration is visible to all actions and may be overridden in actions by defining a local dispatch configuration with the same name.

Attributes

The <dispatches> element has no attributes.

Body

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

```
(dispatch*)
```

Related elements

<dispatch>

3.1.5. The <dispatch> Element

Purpose

The <dispatch> element defines a dispatch configuration.

It may appear either inside the <dispatches> element to define a *global* dispatch configuration or inside an <action> element to define a *local* dispatch configuration.

A <dispatch> element requires either the path or action attribute to specify the dispatch target.

Attributes

Name	Type	Description
name	NMTOKEN	The dispatch name. The name attribute must be unique under <dispatch> siblings. Omitting the name sets the corresponding configuration property to null. Therefore, only one sibling may do this.
path	CDATA	Dynamic - The dispatch path. The interpretation of this attribute may depend on the dispatcher. The default

		dispatcher assumes a context-relative path, if it starts with a slash ('/'). Otherwise, if the redirection flag is set, it assumes an absolute URL. Otherwise, a path relative to the current request path is assumed.
action	CDATA	The dispatch action path. This is the path of the action, this configuration dispatches to. If the module attribute is set, it specifies the name of the module containing the action; otherwise, the current module is assumed. Either the path or action attribute must be set.
module	NMTOKEN	The dispatch action module. See above. The module attribute requires the action attribute to be set.
redirect	true|false	Instruct the dispatcher to use redirection, if set to true. The default is false.
dispatcher	NMTOKEN	Specifies a custom dispatcher to be used.

Body

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

```
(param*)
```

Related elements

<param>, <dispatches>, <action>

3.1.6. The <exception-handlers> Element

Purpose

The <exception-handlers> element is a container for global <exception-handler> elements. A global exception handler configuration is visible to all actions and may be overridden in actions by defining a local exception handler configuration for the same exception type.

Attributes

The <exception-handlers> element has no attributes.

Body

The body of the <exception-handlers> element is defined by the following sequence:

```
(exception-handler*)
```

Related elements

<exception-handler>

3.1.7. The <exception-handler> Element

Purpose

The <exception-handler> element defines an exception handler.

It may appear either inside the <exception-handlers> element to define a *global* exception handler or inside an <action> element to define a *local* exception handler.

Attributes

Name	Type	Description
class	CDATA	Required, Dynamic - The name of a class implementing the Plugin interface.
type	CDATA	The name of the exception class, the exception handler should handle. Defaults to java.lang.Exception.

Body

The body of the <exception-handler> element is defined by the following sequence:

(param*)

Related elements

<param>, <exception-handlers>, <action>

3.1.8. The <actions> Element

Purpose

The <actions> element is a container for <action> elements.

Attributes

The <actions> element has no attributes.

Body

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

(action*)

Related elements

<action>

3.1.9. The <action> Element

Purpose

The <action> element defines an action.

Attributes

Name	Type	Description
path	CDATA	Required - The module-relative action path, starting with a slash ('/'). One action may specify "/*" to be used as default action for requests, whose module-relative path does not match any other action path.
class	CDATA	Dynamic - The name of a class implementing the Action interface. If omitted, the module will use the default action class.
dispatcher	NMTOKEN	Specifies a custom dispatcher to be used. May be overridden by dispatch configurations.

Body

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

```
(param*,filter*,dispatch*,exception-handler*)
```

Related elements

<param>, <filter>, <dispatch>, <exception-handler>, <actions>

3.1.10. The <filter> Element

Purpose

The <filter> element defines an action filter.

Either the class attribute or the name attribute can be used to specify the filter class.

The when attribute may contain an EL expression to conditionally include the filter during action processing.

Attributes

Name	Type	Description
class	CDATA	Dynamic - The name of a class implementing the Filter interface.
name	NMTOKEN	The name, under which a filter class has been registered by a plugin.
when	CDATA	Dynamic - An EL expression (without "${" and "}"). If given, the expression is evaluated to a boolean value during action processing. If it evaluates to false, the module skips the filter when executing the filter chain.

Body

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

```
(param*,dispatch*)
```

Related elements

<param>, <dispatch>, <action>

3.1.11. The <param> Element

Purpose

The <param> element defines a parameter by associating a name with a string value. <param> elements may appear inside <plugin>, <action>, <filter>, <dispatch> and <exception-handler> elements.

Attributes

Name	Type	Description
name	NMTOKEN	Required - The parameter name.
value	CDATA	Required, Dynamic - The parameter value.

Body

The <param> element has no body.

Related elements

<plugin>, <dispatch>, <exception-handler>, <action>, <filter>

3.2. Accessors

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

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

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

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

3.2.1. The control.errors accessor

errors

Answers a `de.odysseus.calyxo.base.Messages` instance containing error messages by delegating to the module's message support.

Example

The expression `${calyxo.control.errors.allMessages}` evaluates to an iterator over the action error messages, that have been saved using the module's message support instance. Thus,

```
<ul>
  <c:forEach var="message" items="${calyxo.control.errors.allMessages}">
    <li>${calyxo.base.i18n.format[message]}</li>
  </c:forEach>
</ul>
```

renders all errors into an unordered list.

3.2.2. The control.warnings accessor

warnings

Answers a `de.odysseus.calyxo.base.Messages` instance containing warning messages by delegating to the module's message support.

Example

The expression `${calyxo.control.warnings.allMessages}` evaluates to an iterator over the action warning messages, that have been saved using the module's message support instance. Thus,

```
<ul>
  <c:forEach var="message" items="${calyxo.control.warnings.allMessages}">
    <li>${calyxo.base.i18n.format[message]}</li>
  </c:forEach>
</ul>
```

renders all warnings into an unordered list.

3.2.3. The control.infos accessor

infos

Answers a `de.odysseus.calyxo.base.Messages` instance containing info messages by delegating to the module's message support.

Example

The expression `${calyxo.control.infos.allMessages}` evaluates to an iterator over the action info messages, that have been saved using the module's message support instance. Thus,

```
<ul>
  <c:forEach var="message" items="${calyxo.control.infos.allMessages}">
    <li>${calyxo.base.i18n.format[message]}</li>
  </c:forEach>
</ul>
```

renders all infos into an unordered list.

4. Miscellaneous

4.1. Adding Groovy Support

Calyxo Control may be easily configured to add support for the Groovy scripting language (requires Groovy JSR-03 or later).

You only have to specify a different class loader to your module context. Just use `groovy.lang.GroovyClassLoader` as the module's class loader and tell it to load groovy classes from `/WEB-INF/groovy` by adding the following to your configuration:

```
<use value="${moduleContext}">
  <property name="classLoader">
    <object class="groovy.lang.GroovyClassLoader">
      <constructor>
        <arg value="${moduleContext.classLoader}"/>
      </constructor>
      <method name="addClasspath">
        <arg>
          <member value="${moduleContext.servletContext}">
            <method name="getRealPath">
              <arg value="/WEB-INF/groovy"/>
            </method>
          </member>
        </arg>
      </method>
    </object>
  </property>
</use>
```

Now the module will search for user defined classes in `/WEB-INF/groovy` before falling back to the default class loading mechanism.

This means that you now can either use Groovy or Java to implement any of your classes that will be loaded by this module!

Groovy Accessors, EL Functions, Plugins, Actions, Filters, ...

5. Project

5.1. History of Changes

Version 0.9.0 (2006/10/28)

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

Version 0.9.0-rc3 (2006/02/12)

developer: **cbe** ***context:*** **code** ***type:*** **add**
Replaced `ControlModuleSupport` by `ControlModuleGroup`. Renamed `ModuleContext`, `Mapping`, `Mappings` to `ControlModuleContext`, `ControlModuleMapping`, ...
developer: **cbe** ***context:*** **code** ***type:*** **update**
Eliminated dependency to `commons-collections`.

Version 0.9.0-rc2 (2005/06/26)

developer: **cbe** ***context:*** **code** ***type:*** **add**
Added setter for `ModuleContext`'s `classLoader` property.

developer: **cbe** ***context:*** **code** ***type:*** **add**
exception-handler elements may now contain dispatch elements to be more consistent with filter elements.
developer: **cbe** ***context:*** **code** ***type:*** **add**
Support conditional filter execution by adding the `when` attribute to `<filter>` element. The attribute contains an EL expression (without '`${`' and '`}`') which is evaluated to a boolean value.

Version 0.9.0-rc1 (2005/03/07)

developer: **cbe** ***context:*** **code** ***type:*** **update**
Changed namespace to `http://calyxo.odysseus.de/xml/ns/control`.

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:*** **docs** ***type:*** **add**
Added documentation.
developer: **cbe** ***context:*** **code** ***type:*** **update**
Major refactoring. Plugin/Action/Filter/ExceptionHandler/Dispatcher are now interfaces.

Version 0.9.0-b1 (2004/06/03)

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

5.2. Todo List

medium priority

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