Handling Server Errors

{scrollbar}

Error handling for MyFaces Core 2.0 and later versions

Since JSF 2.0, it is possible to provide a custom javax.faces.context.ExceptionHandler or javax.faces.context.ExceptionHandlerWrapper implementation to deal with exceptions. To do that, just create your custom class, an factory that wrap/override it and add the following into your faces-config.xml:

faces-config.xml<faces-config xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-facesconfig_2_0.xsd" version="2.0"> <!-- ... --> <factory> <!-- ... --> <exception-handler-factory>org.apache.myfaces.context.ExceptionHandlerFactoryImpl</exception-handler-factory> <!-- ... --> </factory> <!-- ... --> </faces-config>

This is an example of an ExceptionHandlerFactory, from myfaces code:

ExceptionHandlerFactoryImpl.javapublic class ExceptionHandlerFactoryImpl extends ExceptionHandlerFactory { @Override public ExceptionHandler getExceptionHandler() { return new SwitchAjaxExceptionHandlerWrapperImpl(new MyFacesExceptionHandlerWrapperImpl(new ExceptionHandlerImpl()) , new AjaxExceptionHandlerImpl()); } }

If you need a wrapper:

ExceptionHandlerFactoryImpl.javapublic class ExceptionHandlerFactoryImpl extends ExceptionHandlerFactory { @Override public ExceptionHandler getExceptionHandler() { return new CustomExceptionHandlerWrapper(getWrapped().getExceptionHandler()); } }

The most important method to override is `ExceptionHandler.handle()`.

MyFacesExceptionHandlerWrapperImpl.javapublic class MyFacesExceptionHandlerWrapperImpl extends ExceptionHandlerWrapper { //... public void handle() throws FacesException { //... some custom code goes here ... } }

Take a look at MyFaces Core source code, to know in detail how ExceptionHandler implementations works.

MyFaces ExceptionHandler

MyFaces Core provide a custom ExceptionHandler to deal with exceptions and provide detailed information about it. Since 2.0.8/2.1.2 this is disabled on Production environments unless it enabled on web.xml file. Don't forget to provide your custom error page in this scenario, to prevent show more information than necessary.

xml <context-param> <param-name>org.apache.myfaces.ERROR_HANDLING</param-name> <param-value>true</param-value> </context-param> <!-- if you want to use a different resource template file than "META-INF/rsc/myfaces-dev-error.xml" this param let you configure it. (since 1.2.4-SNAPSHOT and 1.1.6-SNAPSHOT)--> <context-param> <param-name>org.apache.myfaces.ERROR_TEMPLATE_RESOURCE</param-name> <param-value>META-INF/rsc/custom-dev-error.xml</param-value> </context-param>

Provide an error page

If org.apache.myfaces.ERROR_HANDLING is set to "false", or if it is undefined and the project stage is "Development", the default ExceptionHandler just throws an exception. So you can still setup an error page by adding something like this in your web.xml file:

xml <error-page> <error-code>500</error-code> <location>/somepage.jsp</location> </error-page>

Error handling for MyFaces Core 1.2 and earlier versions

MyFaces, from version 1.2.1 and 1.1.6, includes automatic error-handling for the full JSF-Lifecycle (taken over mostly from Facelets, with a few adoptions and additions). So for most projects during development, you will have exactly what you want with these new error-handling possibilities.

If this is not what you want, though, you can always disable or modify this error-handling with the following parameters:

xml <!-- if you want to disable the behaviour completely --> <context-param> <param-name>org.apache.myfaces.ERROR_HANDLING</param-name> <param-value>false</param-value> </context-param> <!-- if you are using myfaces + facelets don't forget to do this --> <context-param> <param-name>facelets.DEVELOPMENT</param-name> <param-value>false</param-value> </context-param> <!-- if you want to use a different resource template file than "META-INF/rsc/myfaces-dev-error.xml" this param let you configure it. (since 1.2.4-SNAPSHOT and 1.1.6-SNAPSHOT)--> <context-param> <param-name>org.apache.myfaces.ERROR_TEMPLATE_RESOURCE</param-name> <param-value>META-INF/rsc/custom-dev-error.xml</param-value> </context-param> <!-- if you want to choose a different class for handling the exception - the error-handler needs to include a method handleException(FacesContext fc, Exception ex)--> <context-param> <param-name>org.apache.myfaces.ERROR_HANDLER</param-name> <param-value>my.project.ErrorHandler</param-value> </context-param>

If you do this, you can now read on to get to general ways of handling server-errors.

Server errors such as HTTP 500 can occur for a number of reasons such as uncaught exceptions, missing JSFs or backing beans, bad URL and the list goes on. While we hope these only occur during development it is important to plan to catch and deal with these errors gracefully when running live with multiple users.

Several approaches have been discussed on the mailing list:

Use default handler

Myfaces has a default error handler (class javax.faces.webapp._ErrorPageWriter) that uses a jsp template file (META-INF/rsc/myfaces-dev-error.xml and META-INF/rsc/myfaces-dev-debug.xml) to handle errors.

For define a custom template file:

xml <context-param> <param-name>org.apache.myfaces.ERROR_HANDLING</param-name> <param-value>true</param-value> </context-param> <context-param> <param-name>org.apache.myfaces.ERROR_TEMPLATE_RESOURCE</param-name> <param-value>META-INF/rsc/mycustom-template-error.xml</param-value> </context-param>

Use sandbox org.apache.myfaces.tomahawk.util.ErrorRedirectJSFPageHandler

This handler uses myfaces error handling feature, redirecting to a jsf page when an error occurs.

An example jsf page for redirect can be found at http://issues.apache.org/jira/browse/TOMAHAWK-1297

This class is set as a config-parameter org.apache.myfaces.ERROR_HANDLER
available on myfaces core jsf. (This does not work with RI)

The idea is extends myfaces error handling feature, making possible to redirect
to a jsf page when an error occur, using navigation rules.

If this handler is not able to handle the error, an alternate error handler
could be set in the config-parameter org.apache.myfaces.ERROR_REDIRECT_ALTERNATE_HANDLER

The info of the error in the jsf page can be found using:

#{exceptionContext.cause} : Cause retrieved from the exception #{exceptionContext.stackTrace} : Stack trace of the exception #{exceptionContext.exception} : Exception handled by this page #{exceptionContext.tree} : Print the component tree of the page that cause the error #{exceptionContext.vars} : Enviroment variables from the request

Using servlets

Mert Caliskan (http://www.jroller.com/page/mert?entry=handling_errors_with_an_errror) describes an approach which wraps the JSF servlet with a new servlet which delegates to the faces servlet but handles uncaught exceptions allowing the developer to redirect to a custom error page.

Andrea Paternesi has refined this technique for MyFaces as described here:
[http://patton-prog-tips.blogspot.com/2008/10/myfaces-handling-viewexpiredexception.html]

With JSF

Another approach is described in the book 'Core Server Faces' which uses the servlet engine to catch the error and delegate to a JSF error page. While not as elegant as the first approach this method has several advantages for some users

1 It uses a standard JSP approach and framework<
>
2 It does not require custom servlets<
>
3 It can easily be customized/changed by simply changing the error handler definition in the web.xml<
>

To implement this method perform the following steps

1 define an error handling web page in web.xml

xml <error-page> <error-code>500</error-code> <location>/ErrorDisplay.jsf</location> </error-page>

2 Create the error handler display. Due to a problem with the JSF 1.1 specification, the error handling page cannot use a <f:view> but must use a subview.

xml<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"> <f:subview id="error" xmlns:f="http://java.sun.com/jsf/core" xmlns:t="http://myfaces.apache.org/tomahawk" xmlns:h="http://java.sun.com/jsf/html"> <html> <head> <meta content="no-cache" http-equiv="Cache-Control" /> <meta content="no-cache" http-equiv="Pragma" /> <title>CMS - Error</title> <t:stylesheet path="#{SessionBean.styleSheet}" /> </head> <body> <h:form> : : set up the normal view : <h:outputText styleClass="infoMessage" escape="false" value="#{ErrorDisplay.infoMessage}" /> <t:htmlTag value="br" /> <h:inputTextarea style="width: 99%;" rows="10" readonly="true" value="#{ErrorDisplay.stackTrace}" /> : : more view stuff : </h:form> </body> </html> </f:subview>

3 Create a backing bean in request scope which can process the error. (don't forget to register it on faces-config.xml)

ErrorDisplay.javaimport cms.beans.framework.AbstractUIBean; import com.c2gl.jsf.framework.ApplicationResource; import java.io.PrintWriter; import java.io.StringWriter; import java.util.Map; import javax.faces.context.FacesContext; import javax.servlet.ServletException; public class ErrorDisplay extends AbstractUIBean { private static final long serialVersionUID = 3123969847287207137L; private static final String BEAN_NAME = ErrorDisplay.class.getName(); public String getInfoMessage() { return "An unexpected processing error has occurred. Please cut and paste the following information" + " into an email and send it to " + some email address + ". If this error " + "continues to occur please contact our technical support staff at " + some phone number etc + "."; } public String getStackTrace() { FacesContext context = FacesContext.getCurrentInstance(); Map requestMap = context.getExternalContext().getRequestMap(); Throwable ex = (Throwable) requestMap.get("javax.servlet.error.exception"); StringWriter writer = new StringWriter(); PrintWriter pw = new PrintWriter(writer); fillStackTrace(ex, pw); return writer.toString(); } private void fillStackTrace(Throwable ex, PrintWriter pw) { if (null == ex) { return; } ex.printStackTrace(pw); if (ex instanceof ServletException) { Throwable cause = ((ServletException) ex).getRootCause(); if (null != cause) { pw.println("Root Cause:"); fillStackTrace(cause, pw); } } else { Throwable cause = ex.getCause(); if (null != cause) { pw.println("Cause:"); fillStackTrace(cause, pw); } } } }

Also have a look at our ExceptionUtils class. It encapsulates the way how to get the real root cause

java [1] List exceptions = ExceptionUtils.getExceptions(exception); [2] Throwable throwable = (Throwable) exceptions.get(exceptions.size()-1); [3] String exceptionMessage = ExceptionUtils.getExceptionMessage(exceptions);

1 get a list of all exceptions - using getRootCause if available or getCause<
>
2 get the initial exception<
>
3 get the first exception with an message starting with the initial exception

So the new fillStackTrace become

java private void fillStackTrace(Throwable ex, PrintWriter pw) { if (null == ex) { return; } List exceptions = ExceptionUtils.getExceptions(exception); Throwable throwable = (Throwable) exceptions.get(exceptions.size()-1); for (int i = 0; i<exceptions.size(); i++) { if (i > 0) { pw.println("Cause:"); } throwable. printStackTrace(pw); } }

In the backing bean we construct a message which informs the user that something we didnt't plan for has happened with directions on who to call, what to do etc. We don't really care if they actually do cut-and-paste the error and email it to us as it is also in Tomcat's logs but giving the user something to do and become part of resolving the problem is always a good idea 🙂

ViewExpiredException: No saved view state could be found for the view identifier

In some configurations, you might run into this problem exception with the error handling method shown above. This will happen if an error results in a forward, rather than redirect. The ''ViewHandler'' will call ''response.sendError()'' in case of an error, which will lookup your ''<error-page>'' declarations in ''web.xml'' and forward to the error url. If the error url is picked up by the ''FacesServlet'' (i.e. it's a JSF url), a new JSF lifecycle will be started. However, since this is a forward, the request object will still contain all of the request parameters, including ''"javax.faces.ViewState"'', which makes the request look like a postback to the ''ViewHandler''. Since it's a postback, the ''ViewHandler'' will expect a saved view, which is clearly not going to be there, since our ''viewId'' is now referencing the error page.

This problem can be solved by customizing the ''ViewHandler'' to use ''response.sendRedirect()'' instead of ''response.sendError()'', but that will mean that we can no longer use ''web.xml'' for specifying the error page mappings.

Another way to handle this would be to use an intermediate step by specifying a non-JSF URL as the error page and then somehow redirecting to the JSF error page. For example, for the 404 error code you could specify ''/error/404_redirect.html'':

xml<html><head><meta http-equiv="Refresh" content="0; URL=/DPSV4/error/404.jsf"></head></html>

This works, but requires you to hard code the context path.

The solution I ended up with involves a ''RedirectServlet'':

RedirectServlet.javapublic class RedirectServlet extends HttpServlet { private static final String URL_PREFIX = "url="; @Override protected void doGet (HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException { String query = req.getQueryString(); if (query.contains (URL_PREFIX)) { String url = query.replace(URL_PREFIX, ""); if (!url.startsWith(req.getContextPath())) { url = req.getContextPath() + url; } resp. sendRedirect(url); } } @Override protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException { doGet (req, resp); } }

used like this in ''web.xml'':

xml <servlet> <servlet-name>Redirect Servlet</servlet-name> <servlet-class>ca.gc.agr.ops.web.jsf.redirect.RedirectServlet</servlet-class> </servlet> <servlet-mapping> <servlet-name>Redirect Servlet</servlet-name> <url-pattern>/redirect</url-pattern> </servlet-mapping> <error-page> <error-code>403< /error-code> <location>/redirect?url=/error/403.jsf</location> </error-page> <error-page> <error-code>404</error-code> <location>/redirect?url=/error/404. jsf</location> </error-page> <error-page> <error-code>500</error-code> <location>/redirect?url=/error/500.jsf</location> </error-page>

With plain JSP

If you didn't require any JSF functionality in your JSP page it might be worth to consider using the following error.jsp

[http://svn.apache.org/viewvc/myfaces/tomahawk/trunk/examples/simple/src/main/webapp/error.jsp?view=markup]

with this web.xml configuration

xml <error-page> <error-code>500</error-code> <location>/error.jsp</location> </error-page>

This reduces the number of technologies required to show the error page which might improve the availablity of this page 😉

Custom error handler under Apache Geronimo JavaEE server

If you would use your own error handler (org.apache.myfaces.ERROR_REDIRECT_ALTERNATE_HANDLER) under Apache Geronimo, you must set properly your deployment plan (geronimo-web.xml usualy), otherwise you will have classloading problem of your class. By default the MyFaces classes are loaded to your classpath through dependencies at org.apache.geronimo.framework.jee-specs/CAR. This is OK for common cases, but if you instruct MyFaces to use your own error handler class, you get the error because MyFaces cannot find your class in calling class.forName(). Avoid this situation is quite simple - in your deployment plan specify dependencies on myfaces-api and myfaces-impl and then modify classloading via hidden-classes setting.

A fragment of your dependency plan would be like this:

xml<dep:dependencies> . . <dep:dependency> <dep:groupId>org.apache.myfaces.core</dep:groupId> <dep:artifactId>myfaces-api</dep:artifactId> <dep:type>jar</dep:type> </dep:dependency> <dep:dependency> <dep:groupId>org.apache.myfaces.core</dep:groupId> <dep:artifactId>myfaces-impl</dep:artifactId> <dep:type>jar</dep:type> </dep:dependency> . . </dep:dependencies> <dep:hidden-classes> <dep:filter>javax.faces</dep:filter> <dep:filter>org.apache.myfaces</dep:filter> </dep:hidden-classes>

This solution was tested under Apache Geronimo 2.1.3., but will probably be similar in other versions.{scrollbar}