Techniques for Handling JSF Exceptions, Messages and Contexts

Dwight Deugo
Carleton University, School of Computer Science
1125 Colonel By Drive, Ottawa, Ontario, K1S 5B6 Canada
deugo@scs.carleton.ca

Abstract - JavaServer Faces is a new technology that helps developers build web applications using Java. Rather than generating stack traces as responses to invalid client requests, in this paper we describe techniques for gracefully handling checked and unchecked exceptions and HTML status codes in JavaServer Faces applications. As part of our solution we describe how to create messages for the user as responses to errors. Since it is never advisable to have your business layer know which application framework your user interface layer is using, we also describe a method for making objects and messages from the business layer available to the user interface layer without coupling the former to the latter.

Keywords - JavaServer Face, Exceptions, Status Codes

1. Introduction

JavaServer Faces (JSF) [1, 2, 10] is a technology to help developers build user interfaces (UIs) for web applications using Java. For years, other frameworks such as Struts [12, 3] have aided developers build web applications using a variation of the classic Model-View-Controller design paradigm. Craig McClanahan, Senior Staff Engineer at Sun Microsystems and the originator of Struts, has played a leading role in bring JSF to where it is today. Given McClanahan is at the root of both technologies, it is not surprising the similarities between the two frameworks.

One area that is different between the two frameworks is the way exceptions and messages are handled. In Struts, you can define an ExceptionHandler to execute when a Struts Action method throws an Exception. To make this work you must subclass org.apache.struts.action.ExceptionHandler and override the execute method. The execute method must process the Exception and then return an ActionForward object to tell the framework where to forward to next. Finally, you must configure your handler in struts-config.xml file to look for those exceptions as follows:

```
<global-exceptions>
   <exception
      key="key"
      type="java.io.Exception"
      handler=
            "com.yourPackageName.ExceptionHandler"/>
</global-exceptions>
```

In this case, when a java.io.Exception is received, the execute method in the corresponding ExceptionHandler is invoked.

The reason for going to such lengths to handle exceptions is that you don't want to present users on the client side with stack traces when exceptions occur on the server, such as the one found in Figure 1. Users don't know or care about exceptions. That information is in the domain of the developer. Therefore, it is always advisable to handle exceptions and present clients with web pages that contain information useful to them, such as messages indicating the application or database is unavailable, or to please try again later.

A problem of concern in this paper is that JSF does not provide the notion of a global exception, as in Struts. Going a step further there are three different types of errors that must be handled by a JSF application. These errors include the processing of both checked and unchecked Java

exceptions, and the handling HTML error codes. Each error has the potential, if something is not done, to generate a stack trace like the one shown in Figure 1 to the user.

In JSF, as in Struts, application specific messages can be used to convey information to the user about the status of the application. Error messages rather than exceptions are one technique that can be used to present the user with information about errors that have occurred while processing their request. Since errors often occur in the business layer, messages are typically generated at this level. However, it is not advisable to make the business layer aware of the fact it is using JSF or Struts specific messages. What is required is a technique to allow the business layer to generate messages and make them available to the UI layer independent of the UI framework.

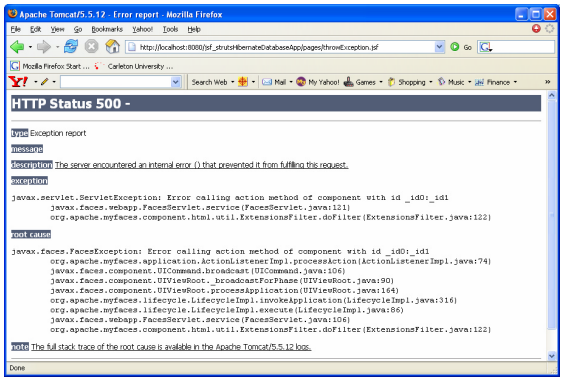

Figure. 1. Stack Trace

In the next section, we provide some background to JSF and exceptions. In section 3 we describe our techniques for handling the different types of exceptions in JSF. In section 4, we provide a technique for isolating messages between the UI and business layers, and in section 5 we summarize.

2. Background

The idea behind JSF is the creation of a standard framework for web application UI components. JSF applications are implemented in Java on the server, and render as web pages back to clients based on their web requests. Sun Microsystems leads this technology as JSR 127 [7]. Major vendors offer tools to support JSF development, including Sun's Java Studio Creator, Borland's JBuilder, IBM's Websphere, Oracle's JDeveloper and Eclipse. Introductory documents are available at Sun's web site describing how to get started with JSF [9].

As a result of being a Java framework, developers working with JSF experience exceptions. When developing a JSF application, developers must handle three different types of exceptions: checked exceptions, unchecked exceptions, and exceptions resulting from HTML status codes. Each exception is a result of a different type of failure.

Checked exceptions represent invalid conditions beyond the immediate control of the Java application, such as invalid user input, database access errors, network problems, or missing files. Java methods must create a policy for all checked exceptions thrown by its implementation. Method can take two different actions. They can surround the offending code with a try/catch block, thus catching the exception and attempting to recover from it. They can avoid the catching of the exception by identifying in the method signature that it can throw a specific exception. The important feature to note here is that in both cases, the developer is made aware of the exception possibility at compile–time and must decide how to handle the situation.

Unchecked exceptions are exceptions representing errors in the program, such as invalid arguments, division by zero, or mistakenly accessing an object bound to a variable that is null. In this case, methods do not need to establish a policy for the handling of an unchecked exception. However, uncheck exceptions often result in stack traces presented to a user either in a web page or in the Java console depending of the Java application type.

There is a controversy over when to use checked and unchecked exceptions [5]. As mentioned in [5]:

> "If a client can reasonably be expected to recover from an exception, make it a checked exception. If a client cannot do anything to recover from the exception, make it an unchecked exception:

In other words, keep unchecked exceptions as exceptional situations, and when possible try to recover from all other exceptions.

HTML status codes are used by the HTTP protocol to indicate the status of a web request. For example, a server will generate a 404 status code if it has not found anything matching the URI used in the request. Like exceptions, these codes need to be handled. The reason is simple. If they are not handled they often result in the generation of an exception when the server side application code is written in Java. In this situation, the default handling mechanism is a stack trace presented to the user.

In all cases it is important that applications not present users with Java stack traces. Applications should fail gracefully alerting the user of the failure situation and giving them guidance on how to continue. Stack traces are developer level information and can be logged and shown to developers for debugging purposes.

3. Exception Techniques

In this section, we describe techniques for handling exceptions and HTML status codes in a JSF application

3.1 Checked Exceptions

For the following technique, the context is as follows:

- You are willing to handle the exception generated in your JSF code.
- You want to indicate to the user that a specific situation has occurred and not show them a stack trace

Handling the exception implies the use of a try/catch block around the offending code. As part of handling the exception, create a FaceMessage, which is provide by the JSF framework, that indicates text of the message you want to present to the users. FacesMessages are creating using the following code.

```
FacesMessage  message = MessageUtils.getMessage(
                        FacesMessage.SEVERITY_ERROR,
                        StringIDToResourceBundleHere,
                        ArrayOfArgumentsOrNullHere);
```

The arguments to the getMessage method include a FacesMessage.Severity, a message id, and an array of Object arguments. There are four different FacesMessage severity levels that are defined in the FacesMessage class:

```
FacesMessage.SEVERITY_INFO
FacesMessage.SEVERITY_WARN
FacesMessage.SEVERITY_ERROR
FacesMessage.SEVERITY_FATAL
```

The message id is a string key pointing into the resource bundle defined by the JSF application. The specific bundle to use is indicated in the JavaServer Page (JSP) displaying the text of the message. A JSF resource bundle is simply a properties file [11] containing key=value pairs and located on the application's classpath. The array of objects contains Strings that are substituted into specific positions within the resulting message.

Keeping the resource Strings separate from the application in a resource bundle helps one to quickly modify the messages without editing the application and enables the String to be internationalized.

To make your message available to the JSP you must add the message to the FacesContext, another class part of the JSF framework. The following code achieves this goal:

```
FacesContext context =
            FacesContext.getCurrentInstance();
context.addMessage(ClientIDStringOrNull,message );
```

To display a message in a JSP, a JSP must first identify the resource bundle. Resource bundles are identified in a JSP by using the JSF f:localBundle tag. For example, the following JSP code identifies the resource bundle MessagesResources.properties under the name "bundle":

```
<f::loadBundle
      basename=MessageResources"
      var="bundle"/>
```

To see all created JSF messages in your JSP, it is simply a matter of using the following messages tag:

```
<h:messages> </h:messages>
```

Other attributes that are part of the message tag syntax are available [13] to display specific messages.

The last part of the approach is to make sure that the correct JSP is used when messages are created as a result of catching exceptions. On approach is to have all of your JSP include the JSF <h:messages> </h:messages> tag to insure messages are always displayed. Another approach is to have your JSF ManagedBeans check for the presence of messages and then provide a navigation String response for any action request that directs to the appropriate JSP to display messages. For example, using the following JSF navigational rule in the Faces configuration file forces any "exception" response from any managed bean action request to display the errorResponse.jsp, which can display the generated messages.

```
<navigation-rule>
      <from-view-id>loginUser.jsp</from-view-id>
      <navigation-case>
         <from-outcome>success</from-outcome>
         <to-view-id>successResponse.jsp</to-view-id>
      </navigation-case>
       <navigation-case>
         <from-outcome>exception </from-outcome>
         <to-view-id>errorResponse.jsp</to-view-id>
      </navigation-case>
   </navigation-rule>
```

Using this technique, control remains with the developer. Catching the exception, generating the messages, navigating to the correct JSP and display the appropriate messages are all under developer control.

3.2 Unchecked Exceptions

For the following technique, the context is as follows:

- You are not willing to handle the exception as a checked exception in your JSF code.
- You want to indicate to the user that a specific situation has occurred and not show them a stack trace

Not handling an exception implies you are not using a try/catch block around the offending code or you did and are throwing an exception. In either case, the effect is that you are passing the exception until a default handler is found that can process the exception. In the case of JSF, the resulting processing produces a web page with a stack trace.

In this technique, the goal is to give you control over these uncaught exceptions and let you navigate to an appropriate JSP to generate the error response. The first step in this technique is to create a new class that implements the javax.faces.event.ActionListener interface, an interface that is part of the JSF framework, e.g.,

```
public class NewActionListener
                    extends ActionListenerImpl
                    implements ActionListener {..}
```

In the case of the Apache MyFaces [10] JSF implementation, the default implementation of the interface is org.apache.myfaces.application.ActionListenerImpl. It is important that your class is a subclass of this class. The reason is that we want the inherited behavior and only want to change how actions are processed.

Next, implement the public void method processAction(ActionEvent event). This method should invoke the "normal" inherited processAction(ActionEvent event) method, catch any exceptions and return the navigational string "exception". Other tasks like logging could also be done when the exception is caught. The goal of the method is to insure that no unchecked exceptions makes it out of the underlying code while processing an action and a navigational response is provided when an exception occurs in order for your application to redirect to a JSP with the appropriate response. The following is an example implementation of the new processAction() method:

```
public void processAction(ActionEvent event) {

    try {
        super.processAction(event);
    } catch (Exception exception) {
        FacesContext facesContext =
              FacesContext.getCurrentInstance();
        Application application =
              facesContext.getApplication();
        NavigationHandler navigationHandler =
             application.getNavigationHandler();
        navigationHandler.handleNavigation(
                          facesContext,
                          null,
```

```
                    "exceptionNavigation");
            facesContext.renderResponse();
        }
}
```

In the above method, when an exception occurs processing a JSF action, the navigational string "exceptionNavigation" is returned.

To force JSF to use your new implementation of the ActionListener you need to modify your Faces configuration (faces-config.xml) file just after the <faces-config> entry. Add the following tag to force JSF to use the new Action Listener you implemented:

```
<application>
    <action-listener>
        com.yourPackageName.NewActionListener
    </action-listener>
</application>
```

You also need to add the global the navigational rule to your faces configuration file that points any "exceptionNavigation" outcome from your new Action Listener to a global error JSP such as exception.jsp.

```
<navigation-rule>
    <navigation-case>
        <from-outcome>
            exceptionNavigation
        </from-outcome>
        <to-view-id>/pages/exception.jsp</to-view-id>
    </navigation-case>
</navigation-rule>
```

The final step to this technique is to implement the error JSP (exception.jsp) to generate the appropriate error response. The error response is up to you. It could be a simple message indicating that things have gone wrong, you might want to display messages if your action listener generated one or more of them, or you might choose to implement something more elaborate giving the user several option links where to go next.

As with the previous technique, exception control remains with the developer. Catching the uncaught exception, generating the messages, navigating to the correct JSP and display the appropriate messages are all under developer control.

3.3 Http Status Code Exceptions

Application behavior resulting in particular HTTP status codes will often cause application servers to show stack traces in resulting web pages. To prevent these stack traces showing up as web pages, add the following to your application's web.xml file to redirect to web pages of your own creation when specific codes are generated.

```
<error-page>
        <error-code>404</error-code>
        <location>/pages/404.html</location>
</error-page>
```

The above entry forces all 404 errors (implying not found, such as a mistyped URL) to respond with the 404.html page. The response page should be an Html page. Do not use a JSP or JSF response page. Using these types of pages forces your application code back into JSF; this will likely generate a reentrant error in processing the response.

Other status codes, such as the 500 error (Internal Server Error), can be handled in the same way.

4. MESSAGE TECHNIQUES

When handling checked exceptions, we described how to create a FacesMessage so that it could be displayed to a user in a response web page. However, exposing objects in the business layer to the FacesMessage class tightly couples that layer to the UI layer. This dependency limits the business layer to being used specifically in and JSF application. Since this situation may not always be the case, what is desirable is a method of exposing the business layer to objects that are important to the UI layer without force the business layer to be dependent on a specific framework.

To solve this problem we begin by creating an IContext interface.

```
import java.util.Iterator;

public interface IContext {

    public static String INFO  = "Info";
    public static String WARN  = "Warn";
    public static String ERROR  = "Error";
    public static String FATAL  = "Fatal";
```

```
    public static String CONTEXT_PROPERTY
                                        = "contextProperty";

    public Iterator getMessagesIterator();

    public void addMessage(
                    String severity,
                    String bundleMessageId,
                    Object[] arguments);

    public Object getProperty(String propertyName);

    public boolean addProperty(
                    String propertyName,
                    Object propertyValue);

    public Object getInternalContext();
}
```

The idea is that classes that implement this interface will support the ability to create specific UI messages, be constructed with specific UI contexts and be able to store other objects, as properties, that can be accessed by the business layer. For example, the following implementation class is a generic context with no internal UI context and support for messages as Strings.

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;

public class Context  implements IContext {

    private static String COLON = ":";
    private ArrayList messages = new ArrayList();
    private Map properties = new HashMap();


    public  Context  () {
        setInternalContext(null);
    }
    public Iterator getMessagesIterator() {
        return getMessages().iterator();
    }
    public void addMessage(
                    String severity,
                    String bundleMessageId,
                    Object[] arguments) {
        getMessages().add(severity +
                        COLON +
                        bundleMessageId);
    }
    private void setMessages(ArrayList messages) {
        this.messages = messages;
    }
    public ArrayList getMessages() {
        return messages;
    }
    protected void setInternalContext(
                        Object internalContext) {
        addProperty(IContext.CONTEXT_PROPERTY,
                    internalContext);
    }
    public Object getInternalContext() {
        return
            getProperty(
                IContext.CONTEXT_PROPERTY);
    }
    public boolean addProperty(String
                        propertyName,
                    Object propertyValue){
        if (getProperties().containsKey(propertyName))
            return false;
        else {
            getProperties().put(
                propertyName,propertyValue);
            return true;
        }
    }
    public Object getProperty(String propertyName) {
        return getProperties().get(propertyName);
    }
    private Map getProperties() {
        return properties;
    }

}
```

The following implementation is a JSF specific implementation, making use of the initial context implementation and specializing to support a FacescContext object internally and the creation of FacesMessages.

```
import javax.faces.application.FacesMessage;
import javax.faces.application.FacesMessage.Severity;
import javax.faces.context.FacesContext;
import org.apache.myfaces.util.MessageUtils;


public class JSFContext extends Context {

    public JSFContext (FacesContext internalContext) {
        setInternalContext(internalContext);
    }
    private  InfoterraContext () {
        setInternalContext(null);
    }
    public void addMessage(
                    String severity,
                    String bundleMessageId,
                    Object[] arguments) {
        getMessages().add(
            MessageUtils.getMessage(


    (Severity)FacesMessage.VALUES_MAP.get(
                            severity),
                        bundleMessageId,
```

```
                    arguments));
    }
}
```

The two constructors forces developers to use the parameterized constructor that requires the FacesContext, part of the JSF framework, as an argument to the implementation. While the FacesContext is not immediately required, it is foreseen that that this object would be useful to have available for the implementation. For example, it would permit developers to not only create FacesMessages internally, but also add them to the FacesContex immediately. This class also benefits from the inherited ability to create properties on the fly.

From either implementation of the IContext, the corresponding class can be initialized in the UI layer, providing specific UI contexts and objects needed by the business layer. The IContext implementation is then passed to the business layer where it can be used to create generic messages and access objects identified by the UI layer that are made available through the property mechanism, such as security or user objects. These ideas are demonstrated below. Managed beans or delegate objects in the UI layer can access MyFaces specific classes as follows:

```
FacesContext context =

     newContext(FacesContext.getCurrentInstance());
context.addProperty("SECURITY", new Security());
```

Objects in the business layer can access only the IContext API as follows:

```
Security user =
     (Security)context.getProperty("SECURITY");
context.addMessage(     IContext.ERROR,
                        "bundleKeyHere",
                        null);
```

5. SUMMARY

Developers are use to seeing Java stack traces. It is all part of development. However, the information in a stack trace is meaningless to an end-user of an application, except to indicate their requests didn't work. Since applications are for users, not for developers, it is important that developers respond to exceptions and errors in their Java application in a meaningful, thought out way and provide users with messages they then can understand.

In this paper we have provided several techniques for handling exceptions and HTML status codes in a JSF application. We also described a method of making messages and UI layer objects available to the business layer that does not tightly couple the two together. For this reason, the IContext interface and its corresponding implementations would also be suitable for use in a Struts application environment too. Checked and unchecked exception and HTML status codes could also be handled in a similar manner, making our contribution wide than just the JSF domain.

6. REFERENCES

[1] D. Geary and C. Horstmann, Core JavaServer Faces, Prentice Hall, 2004.

[2] H. Bergsten, JavaServer Faces, OREILLY, 2001.

[3] J. Holmes, Struts: The Complete Reference, Osborne - McGraw-Hill, 2004.

[4] http://www.dslforum.org/

[5] http://java.sun.com/docs/books/tutorial/essential/exceptions/runtime.html

[6] http://www.w3.org/Protocols/HTTP/HTRESP.html

[7] http://www.jcp.org/en/jsr/detail?id=127

[8] https://javaserverfaces.dev.java.net/

[9] http://java.sun.com/j2ee/javaserverfaces/reference/docs

[10] http://myfaces.apache.org/

[11] http://java.sun.com/docs/books/tutorial/i18n/resbundle/propfile.html

[12] http://struts.apache.org/

[13] http://myfaces.apache.org/tlddoc/core/h/messages.html