Exception Handling in NetUI

Table of contents

1. Introduction

Declarative exception handling is a powerful feature offered by NetUI. It allows you to define handling for expected and unexpected exceptions *without* cluttering your controller logic. Consider the following action method:

```
@Jpf.Action(
    forwards={
        @Jpf.Forward(name="success", path="nextPage.jsp"),
        @Jpf.Forward(name="errorPage1", path="error1.jsp"),
        @Jpf.Forward(name="errorPage2", path="error2.jsp")
    }
)
public Forward someAction()
{
    try
    {
        someBusinessControl.executeSomeBusinessLogic();
        return new Forward("success");
    }
    catch (Exception1 e)
    {
        logger.error("Error 1 occurred while executing business logic.",
e);
        return new Forward("errorPage1");
    }
    catch (Exception2 e)
    {
        logger.error("Error 2 occurred while executing business logic.",
e);
        return new Forward("errorPage2");
    }
}
```

As you can see, the controller logic (executing the business logic and forwarding to "success") is overwhelmed by exception-handling code. The method should really look like this:

```
@Jpf.Action(
    forwards={
        @Jpf.Forward(name="success", path="nextPage.jsp")
    }
)
public Forward someAction() throws Exception1, Exception2
{
    someBusinessControl.executeSomeBusinessLogic();
    return new Forward("success");
}
```

Here, exception handling is left to a mechanism outside of the action method. Read on to find

out how this works.

2. Declaring Exceptions to Handle

The way to declare that a particular exception type should be handled is the `@Jpf.Catch` annotation. Using this annotation, you can specify one of two ways to handle the exception:

- by forwarding to some path, or
- by running an *exception-handler* method.

Here are two `@Jpf.Catch` annotations which deal with an exception type `MyException`; the first one forwards to `error.jsp` and the second invokes an exception-handler method `handleMyException`:

```
@Jpf.Catch(type=MyException.class, path="error.jsp")
@Jpf.Catch(type=MyException.class, method="handleMyException")
```

You can put this annotation in a number of places (in order of precedence):

- Inside a `@Jpf.Action` annotation, which means that the exception is handled only for that action.
- Inside a `@Jpf.SimpleAction` annotation, which means that the exception is handled only for that action.
- Inside a `@Jpf.Controller` annotation, which means that the exception is handled for *any action* in the controller class, or, if this is a page flow controller, for `any page` in the page flow.
- Inside a `@Jpf.Controller` annotation in a *base class controller*, which means that the exception is handled for any action in *any derived* controller. If this is a page flow controller, the exception is handled for any page in the page flow.
- Inside a `@Jpf.Controller` annotation in a shared flow controller that is referenced by the current page flow controller. This means the exception is handled for any action or any page in the current page flow.

Here is an example of a page flow controller class that uses `@Jpf.Catch` at the class level and at the action method level:

```
@Jpf.Controller(
    catches={
        @Jpf.Catch(type=Exception1.class, path="error1.jsp"),
        @Jpf.Catch(type=Exception2.class, path="error2.jsp")
    }
)
public class MyPageFlow extends PageFlowController
{
    @Jpf.Action(
```

```
        forwards={
            @Jpf.Forward(name="success", path="success.jsp")
        }
    )
    public Forward begin() throws Exception1, Exception2
    {
        return new Forward("success");
    }

   @Jpf.Action(
        forwards={
            @Jpf.Forward(name="success", path="success.jsp")
        },
        catches={
            @Jpf.Catch(type=Exception1.class, path="specialErrorPage.jsp")
        }
    )
    public Forward specialAction() throws Exception1, Exception2
    {
        return new Forward("success");
    }
}
```

Throughout this page flow, `Exception1` and `Exception2` are handled by navigating to `error1.jsp` and `error2.jsp`, respectively. When the action `specialAction` runs, it will forward to `specialErrorPage.jsp` if `Exception1` occurs.

Note:

If you have a set of `@Jpf.Catch` annotations, and more than one of them applies to a thrown exception, then the *most specific one* in that set will win. Consider the following:

```
    catches={
        @Jpf.Catch(type=Exception.class, path="error1.jsp"),
        @Jpf.Catch(type=NullPointerException.class, path="error2.jsp"),
    }
```

In this case, if `NullPointerException` is thrown, then `error2.jsp` will be shown.

3. Handling Exceptions with Methods

When you use the `method` attribute on `@Jpf.Catch`, you are signalling that you want an exception to be handled with an *exception-handler* method. To make such a method, you use the `@Jpf.ExceptionHandler` annotation on a method with the following signature:

```
public Forward method-name(exception-type ex, String actionName, String
message, Object formBean)
```

The *exception-type* is the specific type of the exception you want to handle (or any superclass type).

In general, an exception-handler method is much like an action method: it has access to member data in the controller class, can perform logic, and chooses a navigation destination.

Here is an example of handling an exception through a method, which increments a count, logs the error, and navigates back to the current page:

```
@Jpf.Controller(
    catches={
        @Jpf.Catch(type=MyException.class, method="handleMyException")
    }
)
public class MyPageFlow extends PageFlowController
{
    private int myExceptionsCount = 0;

    ...

    @Jpf.ExceptionHandler(
        forwards={
            @Jpf.Forward(name="backToCurrent",
navigateTo=Jpf.NavigateTo.currentPage)
        }
    )
    public Forward handleMyException(MyException e, String actionName,
String message, Object formBean)
    {
        ++myExceptionsCount;
        logger.error("My exception occurred.", e);
        return new Forward("backToCurrent");
    }
}
```

4. Sharing Exception Handling

Exception handling works well with the Page Flow inheritance and Shared Flow features, both of which allow you to share exception handling across multiple page flows. The documentation gives guidelines on when to use each one.

5. Displaying Exception Information

There are two main ways to display exceptions using the NetUI JSP tags:

- the `Exceptions` tag
- the `Error` or `Errors` tags

5.1. The Exceptions tag

The `Exceptions` tag is a simple way to display raw exception information. You can use it to show the exception message and/or the exception stack trace. For example, the following page displays just the exception message:

```
An exception occurred:
    <netui:exceptions showMessage="true" showStackTrace="false"/>
```

5.2. The Error and Errors tags

The `Error` or `Errors` tags are used to display localizable messages for exceptions. By default, you must have a message resource whose name is the *class name of the thrown exception* in order to display the message. Consider the following page flow controller:

```
@Jpf.Controller(
    catches={
        @Jpf.Catch(type=IllegalStateException.class, path="error.jsp")
    },
    messageBundles={
        @Jpf.MessageBundle(bundlePath="example.MyMessages")
    }
)
public class Controller extends PageFlowController
{
    @Jpf.Action
    public Forward begin()
    {
        throw new IllegalStateException("Throwing intentionally...");
    }
}
```

Here, an `IllegalStateException` is thrown in the `begin` action, and caught/forwarded to `error.jsp`, which has the following markup:

```
An error occurred: <netui:errors/>
```

This will display the message *named* **java.lang.IllegalStateException** in `/WEB-INF/classes/example/MyMessages.properties`. You could also use the `netui:error` tag to refer to the specific exception message:

```
An error occurred: <netui:error key="java.lang.IllegalStateException"/>
```

If you want to use your own message key (instead of `java.lang.IllegalStateException`), you use the `messageKey` attribute on `@Jpf.Catch`:

```
@Jpf.Catch(type=IllegalStateException.class, path="error.jsp",
messageKey="myMessageKey")
```

Now, you would see the message under key `myMessageKey` in `MyMessages.properties`.

Finally, you can also specify a hardcoded message, or a JSP 2.0-style expression, instead of a message key.

```
@Jpf.Controller(
    catches={
        @Jpf.Catch(type=IllegalStateException.class, path="error.jsp",
message="This is a hardcoded message."),
        @Jpf.Catch(type=NullPointerException.class, path="error.jsp",
message="${pageFlow.class.name}")
    }
)
```

In the first case, the error tags would display the string "This is a hardcoded message". In the second case, they would display the class name of the current page flow controller.