Sakai Errors

Feb 8, 2006

Please direct questions or comments about this document to:
Glenn R. Golden, ggolden@umich.edu

Errors. We would like there to be none, but, alas, sometimes even Sakai users experience errors. We don't always write such perfect code, and things don't always go as planned.

When errors occur, the Sakai kernel, portals, utility layers and applications all work together to detect the problem, minimize damage, inform the Sakai production staff and developers, and finally to inform the end-user.

In Java, errors are usually handled by the exception mechanism. At the point of detection, code will throw an exception. The call stack then starts to unwrap, and code along the way back deal with, clean up from, or ignore the exception. At some point before we run out of call stack, somebody handles the exception in some way.

For the most part, Sakai is request driven (although there are some background tasks that run). The granularity of an error event then is the request. A request can be processed to its successful conclusion, or will be interrupted by an exception and fail in some way.

Java provides two types of exceptions; checked and un-checked. Checked exceptions are part of the various APIs we use – they are declared as possible outcomes of method calls. As such, they are as important as the method parameters, and callers must specifically catch them after the call or pass them on as part of their own API.

Un-checked exceptions are not mentioned in a method's API. These are for things like using null pointers and running out of memory. They might happen anywhere.

The less than idea state of error handling

Up through Sakai 2.1.1, we were not doing such a great job of error handling.

Error detection code in too many places simply ignored the errors – a log entry might be written, but otherwise the error detection was lost too soon.

Errors that were not lost in that way tended to end with the "white screen of death" for the end-user; rather than getting a html response from their request, they get nothing and no indication about what might have gone wrong.

Improved error handling

Sakai now has much better error handling. Low levels of the code that ignored exceptions are now letting them flow. The portal now traps any failure of a request and has elaborate error handling; messages are logged and emailed to support staff; the end-user is informed with an html display, from which feedback can be submitted.

For the new error handling to succeed, the framework, utility code, and applications all have to do the right things when errors occur. There's still work to do in this area, but we are in much better shape now than before.

Error detection

As an application is processing a request, something might go wrong. We must assure that we have good error checking code in Sakai to check for problems. It is possible that some problems can be handled, leading still to a successful request processing. For this, we need to have code that is tolerant of conditions that we can anticipate and deal with. But for many problems we can detect, there will be no solution, and we will decide that the request should fail.

To report an error that we cannot handle, our code needs to throw exceptions. For cases that we can anticipate, we might design into our APIs checked exceptions that we can throw and our callers must catch and deal with. But for other situations, there is a general exception we can throw, or use to wrap an exception we caught. The "ToolException" is defined in the kernel in the ActiveToolComponent API, and can be used to indicate that the tool has given up its attempt to process the request.

Depending on how you design your APIs, it might make sense to declare your own, application-specific exceptions, or declare the ToolException. In many cases it will not. If a method that otherwise does not throw an exception detects an error, we can use unchecked exceptions, the general "RuntimeException" for instance, to throw. This can also be used to wrap any exceptions a method might catch to re-throw so that the error processing can continue.

Minimizing damage

As soon as an error is detected at a low level in the code, and an exception is thrown, the call stack starts to unwind. All the methods are given a chance to deal with the error.

In the bad old Sakai, this is where we lost most of our exceptions. Code was written in try / catch blocks to not let the exceptions flow. This was good, in that it recognized the possibility for failure below, and bad, in that it stopped the processing.

What we want the Sakai code to do is to be aware that errors might occur. It's the responsibility of each method in the call stack to make sure that when the request is interrupted by an error, we cleanup to minimize damage to the Sakai JVM resources and information state. This can be accomplished with try / finally blocks, with appropriate cleanup code in the finally blocks to deal with interrupted processing from an exception.

If a shared critical resource is held (such as a database connection), release it and clean up in the finally block.

Catch blocks can be used to detect non-fatal errors, but otherwise let the exceptions flow.

Error reporting

If all goes well in the call stack, an exception will flow all the way back to the top, which for Sakai, is the portal. (Actually, the request filter is our very top of the request processing call stack, but error handling responsibility is given to the portal).

The Sakai portals now detect any exceptions, and attempt to complete the interrupted request with an error display html for the end-user. The two active portals, Charon and Mercury, are both setup for this now. They use a common helper class, the ErrorReporter, in the util/portal package of Sakai.

Error reporting has three aspects; logging, bug report email, and end-user UI.

Error logging

Sakai's current design makes use of the Servlet API for tool (and helper tool) invocation. We "cross" the Servlet API (i.e. invoke a servlet) many times to handle a single request.

The Servlet methods (doGet, doPost) are declared to throw IOException and ServletException. Our new ToolException is a ServletException, so it naturally flows across the Servlet API.

The standard Servlet implementation does a stack-trace log of any exception that it finds from doGet or doPost, and then re-throws these so they continue to flow. As a consequence, any exceptions heading to the portal will be logged in our "catalina,out" Tomcat logs at least once, perhaps many times (once for each servlet involved).

Here's an example of the Servlet's log (slightly sniped for brevity):

```
ERROR: Servlet.service() for servlet sakai.announcements threw exception
(2006-02-08 10:43:12,844 http-8080-
Processor25_org.apache.catalina.core.ContainerBase.[Catalina].[localhost]
.[/sakai-legacy-tools].[sakai.announcements])
java.lang.reflect.InvocationTargetException
```

```
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke
(NativeMethodAccessorImpl.java:39)
<snip>
        at java.lang.Thread.run(Thread.java:552)
Caused by: java.lang.NullPointerException
        at
org.sakaiproject.tool.announcement.AnnouncementAction.buildMainPanelConte
xt(AnnouncementAction.java:997)
        ... 43 more
```

At the point of detection, it is also a good idea to issue a "WARN" log entry to describe what happened (although the stack-trace might be overkill here). Additional information about the failure can be logged at this point that is not available in the stack-trace.

When the exception rises to the level of the Portal, the root cause exception stack-trace will be logged, along with the user id, usage session id and time stamp:

```
WARN: Error Report from user: null usage session id: null time: Feb 8,
2006 10:43 AM EST userReport: null problem:
org.sakaiproject.api.kernel.tool.ToolException
    at org.sakaiproject.cheftool.ToolServlet.doGet(ToolServlet.java:158)
caused by: java.lang.reflect.InvocationTargetException
    at sun.reflect.NativeMethodAccessorImpl.invoke
(NativeMethodAccessorImpl.java:39)
caused by: java.lang.NullPointerException
    at
org.sakaiproject.tool.announcement.AnnouncementAction.buildMainPanelConte
xt (AnnouncementAction.java:997)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
<snip>
    at java.lang.Thread.run(Thread.java:552)
 (2006-02-08 10:43:12,861 http-8080-
Processor25_org.sakaiproject.util.portal.ErrorReporter)
```

If the end-user chooses to submit a bug report, the bug report details, including the stack-trace, user id, usage session id, time stamp, and user comments, will be logged again, when the report is posted.

Error email

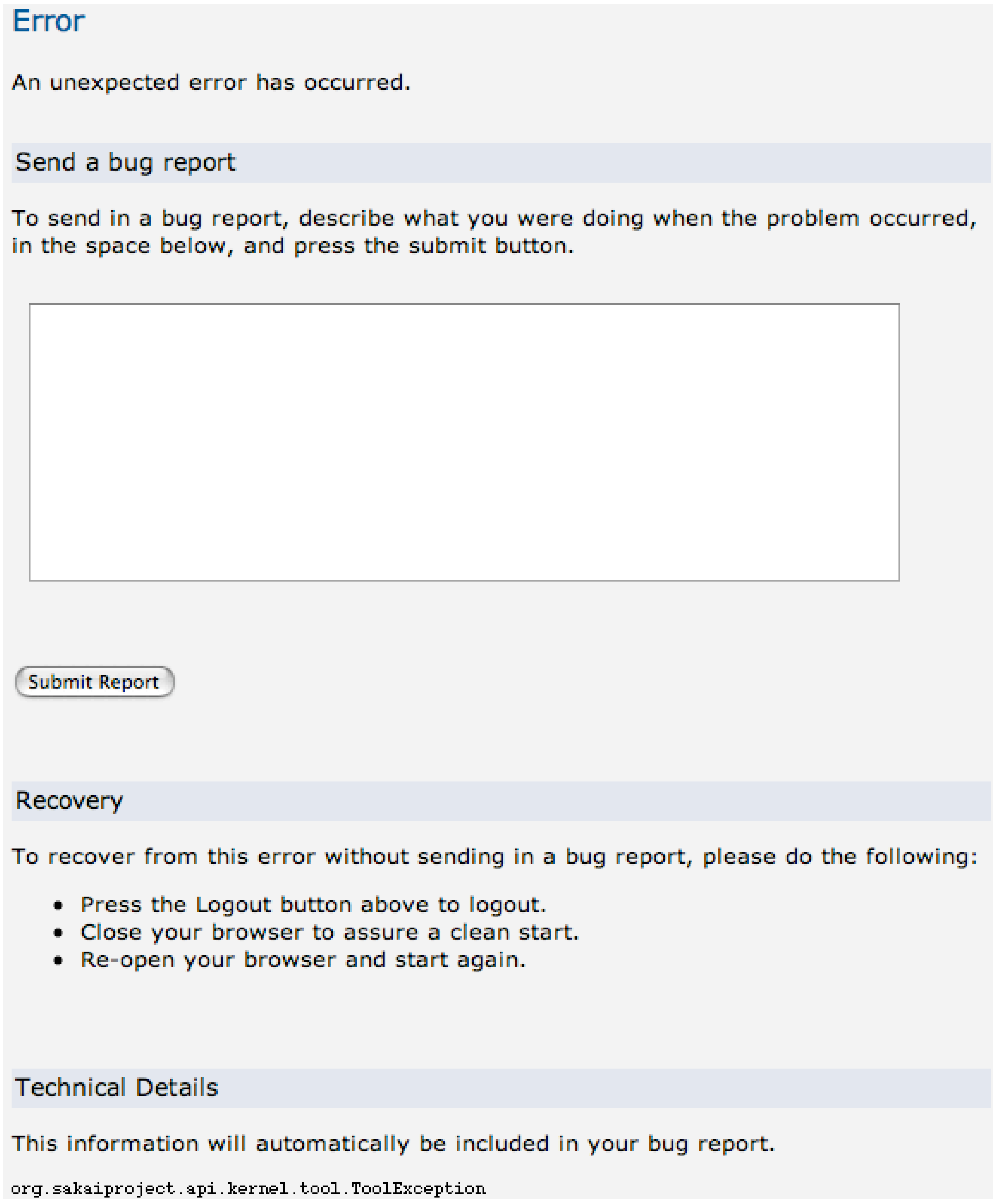
You can configure Sakai with an email address for sending portal-caught errors. In sakai.properties, set:

```
# email address to send errors caught by the portal
portal.error.email=somewhere@abc.edu
```

The email will be from the "no-reply" address of your Sakai server. The subject will be "bug report" with the usage session id (this can be used to more easily group multiple reports from the same usage session). The body will include the stack trace, user id, usage session id and time stamp.

User report

When the portal catches an error, it responds with an error HTML display:



Error

An unexpected error has occurred.

Send a bug report

To send in a bug report, describe what you were doing when the problem occurred, in the space below, and press the submit button.

Recovery

To recover from this error without sending in a bug report, please do the following:

- Press the Logout button above to logout.
- Close your browser to assure a clean start.
- Re-open your browser and start again.

Technical Details

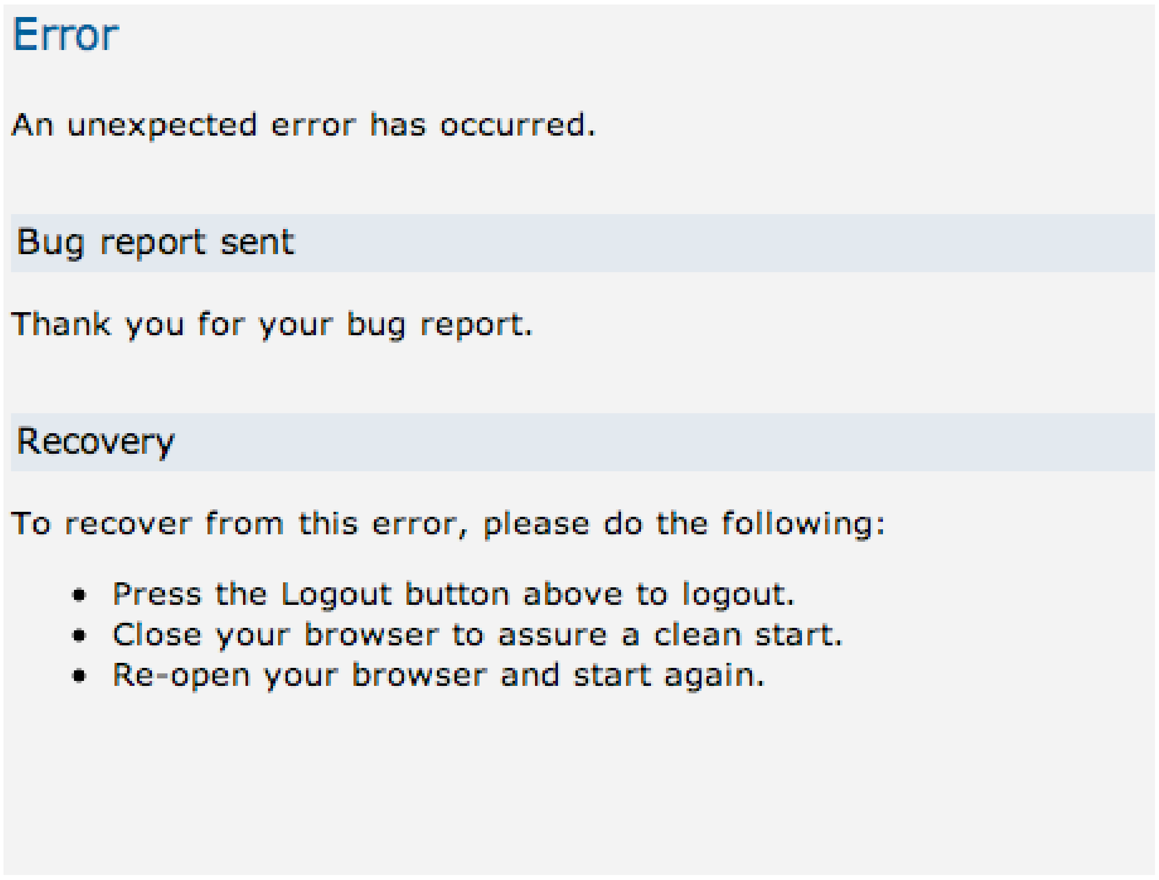
This information will automatically be included in your bug report.

```
org.sakaiproject.api.kernel.tool.ToolException
```

The user can then fill in the text field and submit a bug report, or follow the “Recovery” instructions and continue working.

This display will appear in place of the tool's response, in the same window or iframe.

If the user submits a bug report, it will be logged, optionally emailed, then the user will see this display:



Error

An unexpected error has occurred.

Bug report sent

Thank you for your bug report.

Recovery

To recover from this error, please do the following:

- Press the Logout button above to logout.
- Close your browser to assure a clean start.
- Re-open your browser and start again.

Portal support

To support error detection in a Sakai Portal, you need to catch any throwable that comes up from a tool invocation. The code to do the display formatting, user bug report processing, logging and email is in the ErrorReporter utility class. The portal needs the error detection code, which invokes the report() method of ErrorReporter.

The support user bug report processing, the portal of record must support the "/error-report" and "/error-reported" URL patterns. This is the portal that is configured in sakai.properties, and is the Charon portal for Sakai as configured in distributions. Even if the Mercury portal or some other portal detects the error and initiates the error reporting process, Charon will field user bug reports and respond with the final UI display.

If you are using a different primary portal in Sakai, make sure it supports error reporting. Use the Charon code as an example; support the "/error-report" URL in doPost()

handling, and the “/error-reported” URL in doGet() handling (as well as basic error detection leading to a call to ErrorReporter.report()).