YmtdGeneration

Generation?

JSP touts as an advantage that it takes an existing .jsp page and compiles that into a Servlet for speed and efficiency reasons. What this means is that it first parses the .jsp page into the resulting mess below and then it uses javac or your favorite Java compiler (ie: Jikes) to compile that Servlet into a .class file which is then loaded by the Servlet Engine. Wow, just explaining all of that gave me a headache, how about you?

The point being that using JSP is now a multi step process. The authors of JSP have done a good job of hiding this process stuff behind the scenes in such a way that you do not even notice it. One simply edits the .jsp page in their favorite editor and then uses their browser to call the page via a URI which starts the process of transforming it into a .class file.

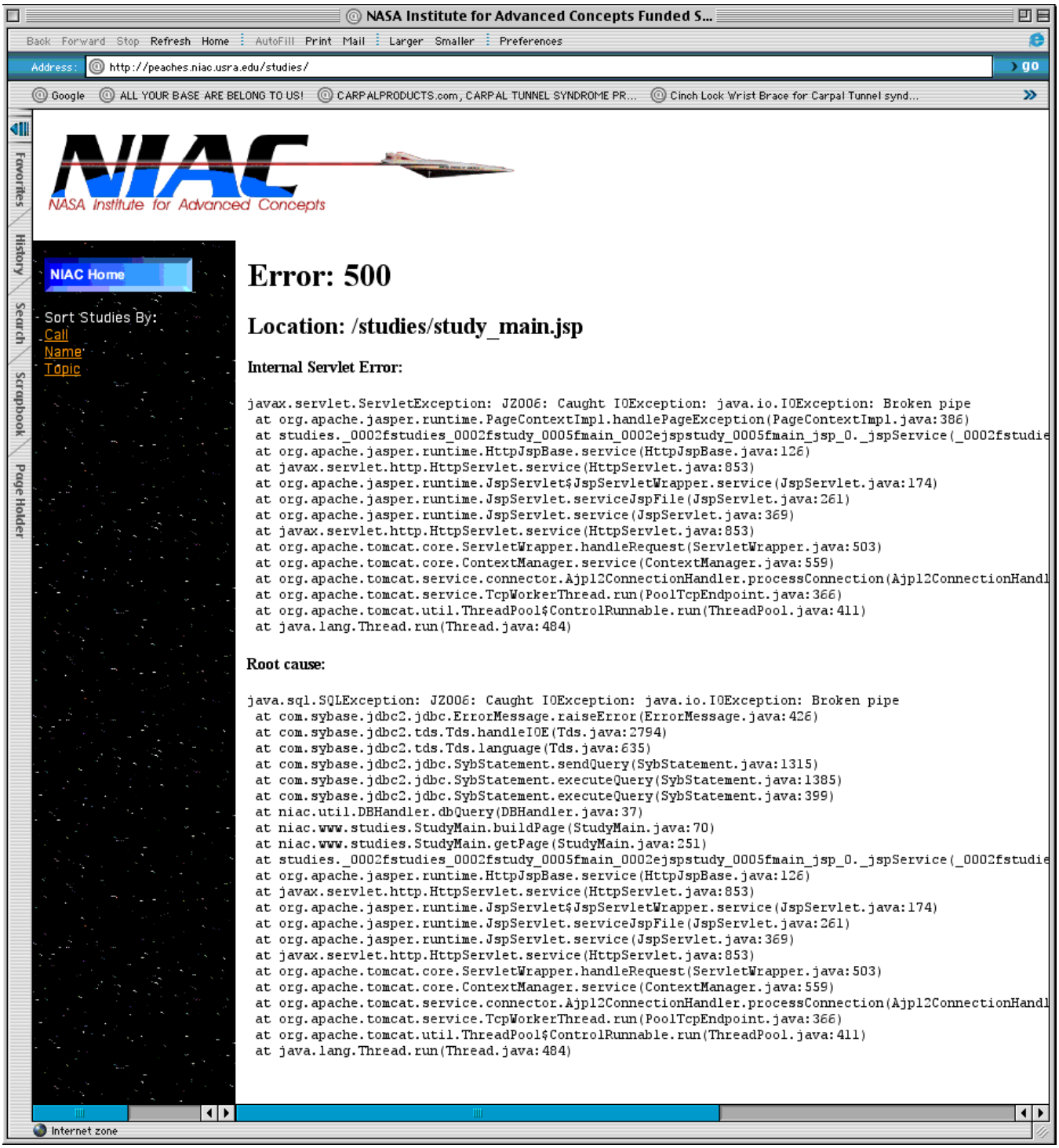
There are some fundamental issues that are being dealt with in the generated .jsp template. The first one is the class name. What happens is that the engine needs to produce a name that is unique in order to work around class loader issues that might crop up. Therefore, each and every time one modifies a .jsp page, a new file is created on disk in the temporary directory. Unfortunately, this directory ends up growing in size until someone decides to clean it up. The engine could probably do this for you, except then it might make the mistake of actually removing the wrong file.

The point being that this whole process of edit->transform->compile->load->run is really unnecessary and in particular, a bad design. On the other hand, Velocity will simply load templates, parse them a single time and then store an Abstract Syntax Tree (AST) representation of the template in memory which can then be re-used over and over again. The process is simply edit->parse->run. The benefit is that working with Velocity templates ends up being much faster and it also removes the requirement of having a javac compiler and temporary scratch directory hanging around. In Velocity, when the template changes, the existing cached template is simply replaced with a freshly parsed version.

Another advantage to Velocity's approach for templates is that the actual template data can be stored anywhere, including a database or remote URI. By using configurable template loaders, it is possible to create template loaders that can do anything that you want.

Even without Turbine, Velocity offers several ways to deal with errors. Where frameworks such as Struts and Turbine come handy is by providing ways of properly dealing with errors. However, due to the fact that Struts is based on top of JSP, it also inherits the same amount of problems associated with JSP. The next chapter will go into more details on that.

One final problem in the design shown below is that the JSP page only catches `Exception`'s. What if the code within a JSP page throws another exception like `OutOfMemoryError`? The problem here is that OOME is based on `Throwable`, not `Exception`. Therefore, it is much more difficult to catch this exception with just a JSP page. Future versions of the JSP spec and implementations will improve on this.



This nice example provided by our friends at NASA, which sends multi-billion dollar equipment through the heavens, is a perfect example of why JSP needs better error handling.

Buffering is also another big issue as constantly writing to the output stream is not very efficient.

```
<%@ page buffer="12kb" %>
<%@ page autoFlush="true" %>
```

These are examples of telling JSP to buffer the output 12kb and to autoFlush the page. Struts+JSP has implemented the MVC model by providing the View portion through JSP templates. What part of the MVC model do you think that those tags belong? You guessed it, not the part where they are being used.

Velocity's approach to dealing with this issue is by allowing the developer to pass a stream into the rendering engine. If there is an exception thrown, during rendering, then the exception can be caught and dealt with. Buffering is also handled by passing properly buffered stream to the parser. Again, if there is an error, then another stream can be easily substituted for the output.

Here is an example of the intermediate code generated by Tomcat 3.3m2:

```
package jsp;

import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class helloworld_1 extends org.apache.jasper.runtime.HttpJspBase {

    static {
    }
    public helloworld_1( ) {
    }

    private static boolean _jspx_inited = false;

    public final void _jspx_init() throws org.apache.jasper.JasperException {
    }

    public void _jspService(HttpServletRequest request, HttpServletResponse  response)
        throws java.io.IOException, ServletException {

        JspFactory _jspxFactory = null;
        PageContext pageContext = null;
        HttpSession session = null;
        ServletContext application = null;
        ServletConfig config = null;
        JspWriter out = null;
        Object page = this;
        String  _value = null;
        try {

            if (_jspx_inited == false) {
                _jspx_init();
                _jspx_inited = true;
            }
            _jspxFactory = JspFactory.getDefaultFactory();
            response.setContentType("text/html;charset=8859_1");
            pageContext = _jspxFactory.getPageContext(this, request, response,
                        "", true, 8192, true);

            application = pageContext.getServletContext();
            config = pageContext.getServletConfig();
            session = pageContext.getSession();
            out = pageContext.getOut();

            out.write("<html>\r\n<head><title>Hello</title></head>\r\n<body>\r\n<h1>\r\n");
            if (request.getParameter("name") == null)
              out.write("\r\n          Hello World\r\n");
            else
              out.write("\r\n          Hello, ");
            request.getParameter("name");
              out.write("\r\n</h1>\r\n</body></html>\r\n");

        } catch (Exception ex) {
            if (out != null && out.getBufferSize() != 0)
                out.clearBuffer();
            if (pageContext != null) pageContext.handlePageException(ex);
        } finally {
            if (out instanceof org.apache.jasper.runtime.JspWriterImpl) {
                ((org.apache.jasper.runtime.JspWriterImpl)out).flushBuffer();
            }
            if (_jspxFactory != null) _jspxFactory.releasePageContext(pageContext);
        }
    }
}
```

You make the decision.