PSU Struts Demo

Table of Contents

Introduction

We've created a simple demonstration of the Apache Jakarta Struts Framework. It aims to address two issues:

- to convince us that Struts is a useful foundation for the next-generation GUS front-end and
- to provide a initial base (eg directory structure, build mechanism) for further development if we decide to go this route.

The demonstration is very simple –

- A front-page: to allow you to switch styles
- A query page
- Query results page

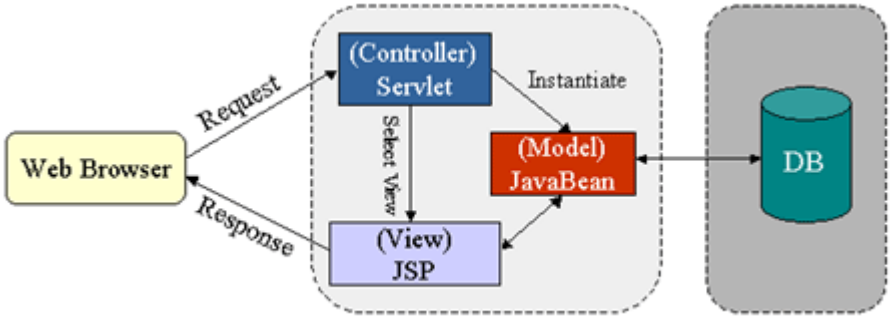
- An alternative query page which demonstrates the Validator mechanism

Each of these is presented with three different look'n'feels (OK looks). These looks mimic the PlasmoDB, Allgenes and GeneDB sites.

The appendix provides an introduction to server-side Java. If you're unfamiliar with servlets and JSPs this is probably the best place to start. Then coming back here we have an architectural overview followed by a series of technical topics. Finally we have a conclusion and a possible action plan.

General Architecture

In the early drafts of the JSP specification (no longer available) Sun outlined how JSPs could be used. The first system is based purely on JSPs. This “Model 1” architecture is simple but hard to maintain and scale. At the other end is a full J2EE enterprise system which is probably overkill for our systems. In the middle is the “Model 2” architecture. In this case requests from the browser are routed through to a servlet. It's the job of the servlet to process the request, eg access a database or run an analysis program. It packages any results in an easy to access form and works out which page to forward these results to. This page then displays the results.



The object(s) that are passed from to the page to be displayed can be any type of Java object. Strings are obviously useful and there is special support for Javabeans in JSP pages. But they could easily be objects from the GUS object layer, SQL blobs, just about anything.

As far as the controller servlet is concerned you have two main choices. You can either have one servlet per function, or one central servlet that does it all. Struts chooses the latter. To avoid having one monolithic, unmaintainable piece of code the Struts controller is mainly responsible for controlling the flow of control between actions and pages. (By default – almost all of the Struts framework is pluggable so you can add to/change any individual part of it.). The domain specific code is implemented in actions. There is more on this later in this document (or possibly earlier if I keep us this rewriting!)

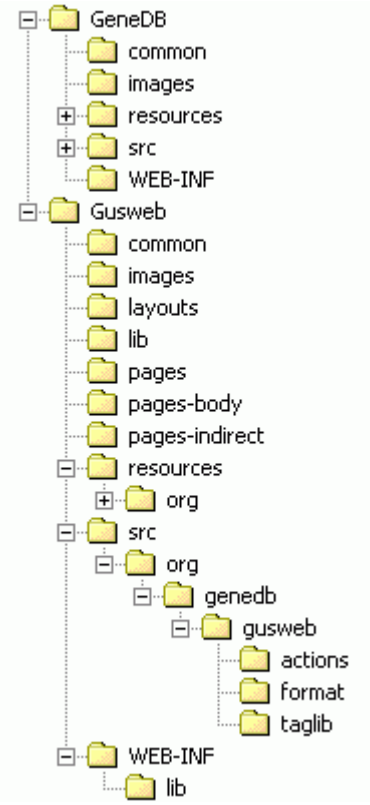
As far as programmatic access is concerned, this architecture easily suports REST-style interactions. SOAP-style web services aren't considered as part of this architecture.

General Architecture (physical)

For the demo we used the Ant build tool. At the top level there is a gusweb directory. If you just deploy this the idea is you get a functional if plain website. Someone adopting GUS can start from here. All of the shared website code/pages goes into gusweb.

There will also normally be a project directory eg GeneDB. The build file overlays the project directory onto gusweb. In this way an individual project can both supplement and override the basic appearance/behaviour.

It's important to note that this development structure doesn't match the deployment structure. The development structure is designed for the developers to be easy to use. The deployment structure is in part constrained by the structure of a web application, as well as the general design principle of making only the bare minimum available to the outside world. For example, for the webapp the src doesn't need to be available and any resources not intended to be accessed directly by the user-agent should be moved under the WEB-INF directory.



In this figure gusweb has ten subdirectories:

- **common**
  For page fragments common between all or most of the pages on a site eg headers and footers
- **images**
  Default images for the gusweb web application.
- **layouts**
  Contains the page layout templates– just one in the demo application
- **lib**
  Is for libraries that should be on the compile-time classpath rather then the runtime one eg

servlet.jar

- **pages**
  Contains the JSP pages that can be accessed directly eg help pages
- **pages-indirect**
  has pages that should only be accessed via a servlet eg query-results page
- **pages-body**
  Both of the above are just simple wrappers around the real content which is stored in this directory
- **resources**
  eg property files
- **src**
  Source files for the common functionality eg searching, shared tag libraries etc
- **WEB-INF**
  The contents in this directory are copied direct to the web application. This contains the web.xml file, tag library definitions, struts configuration files etc. It also contains a lib subdirectory for the runtime libraries. At deployment the gusweb/project class files are also copied in here.

As mentioned each individual project can effectively modify/add to the default gusweb. So GeneDB might have additional layout(s) eg for its organisms' homepages.

Tiles

The three different appearances are achieved quite easily using tiles. Each page says it uses a definition. This defines the page structure in terms of which sections it contains. The demo has the following layout:

```
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html"%>
<%@ taglib uri="/WEB-INF/struts-tiles.tld" prefix="tiles"%>

<html:html>
  <head>
     <title>Struts demo</title>
  </head>
  <body topmargin="0" leftmargin="0" bgcolor="#FFFFFF">
    <tiles:insert attribute="header"/>
    <tiles:insert attribute="message"/>
    <tiles:insert attribute="navBar"/>
    <tiles:insert attribute="bodyContent"/>
    <tiles:insert attribute="motd"/>
    <tiles:insert attribute="footer"/>
  </body>
</html:html>
```

This corresponds to this simple layout:

Header	eg logos
Message	eg Thank you for your email
NavBar	In the demo these are all hard-coded but a custom tag here would have enough knowledge to be context-sensitive
BodyContent	The main contents
Motd	eg Maintenence warning
Footer	eg contact details

Most of the content of these sections is the same on each page. To avoid repeating these details we can set up a mapping file eg tiles-defs.jsp:

```
<%@ taglib uri="/WEB-INF/struts-tiles.tld" prefix="tiles" %>

<tiles:definition
    id="gusweb.default"
    page="/WEB-INF/layouts/guswebDefaultLayout.jsp"
    scope="request">
        <tiles:put name="header" value="/WEB-INF/common/header.jspf" />
        <tiles:put name="message" value="/WEB-INF/common/message.jspf" />
        <tiles:put name="navBar" value="/WEB-INF/common/navBar.jspf" />
        <tiles:put name="motd" value="/WEB-INF/common/motd.jspf" />
        <tiles:put name="footer" value="/WEB-INF/common/footer.jspf" />
</tiles:definition>
```

This sets up the default contents for all the sections except the bodycontent. To use it a page just has to include this file. The front page is simply:

```
<%@ taglib uri="/WEB-INF/struts-tiles.tld" prefix="tiles" %>
<%@ taglib  uri="/WEB-INF/struts-html.tld" prefix="html" %>

<%@include file="/WEB-INF/tiles-defs.jsp" %>

<tiles:insert beanName="gusweb.default" beanScope="request" flush="true">
    <tiles:put name="bodyContent" value="/WEB-INF/pages-body/frontPage-body.
jspf" />
</tiles:insert>
```

The real content of the front page is in frontPage-body.jspf. There is a slight oddity here. You seem to need two pages for every real page using tiles like this – one to say what the content should be and another for the real content. There's probably a way around this...

Actions

Actions capture the domain specific code in a Struts application. An action is in fact subclassed indirectly from HttpServlet. It performs the same basic role. It just has additional hooks and infrastructure as part of the Struts framework. For example the main method (execute) has an ActionForward return value rather than void. The framework uses this to work out where to pass control to next. The QueryAction in the demo is very simple but provides a good demonstration. It's pretty much the same code as you'd write as the body of a normal servlet. (The processQuery method is obviously for the demo only)

```
public ActionForward execute(ActionMapping mapping,
                             ActionForm form,
```

```
                                HttpServletRequest request,
                                HttpServletResponse response)
                        throws Exception {

 DynaActionForm daform = (DynaActionForm) form;

 // Get the 2 values we expect to be set for every query (and need now)
 String organism = (String) daform.get(IConstants.ORGANISM_KEY);
 String queryType = (String) daform.get(IConstants.QUERYTYPE_KEY);

 // Get a DataSource and a connection and pass control to processQuery
 Connection connection = null;
 Result result = null;
 DataSource dataSource = getDataSource(request);
 try {
   connection = dataSource.getConnection();
   result = processQuery(connection, queryType, organism, daform);
 }
 catch (SQLException exp) {
   // If we get an SQL exception go to a more system error type page
   return mapping.findForward(IConstants.FAILURE_KEY);
 }
 finally {
   connection.close();
 }

 // If no results forward to a particular page eg a help page
 if (result.getRowCount() == 0) {
   return mapping.findForward(IConstants.NORESULTS_KEY);
 }

 // Otherwise, set up values for the results page to display
 request.setAttribute(IConstants.RESULTS_KEY, result);
 request.setAttribute(IConstants.STARTROW_KEY,
                      (Integer) daform.get(IConstants.STARTROW_KEY));
 request.setAttribute(IConstants.NBROWS_KEY,
                      (Integer) daform.get(IConstants.NBROWS_KEY));

 return mapping.findForward(IConstants.SUCCESS_KEY);
}


// This looks up the raw SQL for a named query, substitutes in the values
// from the form, submits it to the db and returns the Result
//
private Result processQuery(Connection connection, String queryType,
                            String organism, DynaActionForm daform)
               throws SQLException {

 String sql = null;

...

 // 3 argument queries
 if ("gbTMM".equals(queryType)) {
   Integer minNumTMM = (Integer) daform.get(IConstants.MIN_NBTMM_KEY);
   Integer maxNumTMM = (Integer) daform.get(IConstants.MAX_NBTMM_KEY);
   sql = process3Args("gbTMM", organism, minNumTMM, maxNumTMM);
 }

 Statement stmt = connection.createStatement();
 ResultSet rs  = stmt.executeQuery(sql);

 return ResultSupport.toResult(rs);
}
```

Validation

Struts provides a built-in form validation mechanism. This could be quite useful for feedback forms, surveys etc. For the main querying section it might be useful in the long term. It does allow dynamic form structures to be validated. In the short term it's probably more economical to reuse the existing code.

Database Connection Pooling

All of the major commercial and most of the open-source RDBMS support the JDBC2 connection pooling mechanism. Struts allows this to be centrally configured in the struts-config.xml file. In fact in most modern servlet engines this can be defined for the whole engine). Jakarta Commons also has a generic pooling mechanism.

Logging

Struts uses the Jakarta Common logging layer. This is a thin abstraction over a range of other logging APIs, in particular log4j and the native JDK1.4 logging mechanism.

Configuration Files

In this design we imagine having the configuraton split between a number of files, one per topic, rather than one large file.

Struts ships with the Apache Jackarta Commons Digester which is (yet another!) tool to create and configure (trees of) JavaBeans. Struts uses this for parsing all its configuration files. We haven't played with this at this end for the purposes of this demonstration but it looks easy to use (OnJava article – http://www.onjava.com/pub/a/onjava/2002/10/23/digester.html).

While most of the configuration files will probably be XML-based, we shouldn't veto simple property files where they make sense. If we want to store XML or HTML fragments they may be easier to manage in property files (due to escaping issues.)

Querying

We really like the existing boolean query interface. These are just some discussion points in no particular order, mainly based on local modifications or requests. It presumes the changes previously discussed that languages other than Java should be able to store query results.

- Readable query language: The boolean query code already passes the query information back and forth in the URL. It's proposed to change this to a more human readable format. This would make it easier to debug. Additionally curators could enter complex queries directly rather than building them up via forms, eg: change

  ```
  r:0:pm0=1&r:1:1:pm0=2&r:1:1:pm1=10&query=A:(L:gbChrom,O:(L:gbSigP,L:gbTM))
  ```

  to

```
(gbChrom{chr='1'} AND (gbSigP{} OR gbTM{minTM='2',maxTM='10'})
```

- A complex query can generate a lot of subqueries. Our curators have requested a way for these subqueries not to (always) appear in a user's query history as they can be confusing. A hidden flag in the APIs and results table would enable this.
- We'd also like the interface to remember the current settings when you click on the AND or OR buttons rather than resetting as they do currently.
- The main changes at this end were to make the queries organism specific. As previously explained this meant adding a ParamFactory amongst other changes. We'd like to feed this back into the core searching code. It is worth generalising this with a QueryContext of some description, or just treat organism as a special case?
- We'd like a new SQLEnumParam that talkes both its labels and values from a single query as na small optimisation.

Miscellaneous

Following on from having multiple config file, and precompiling JSP pages it may be worth specifying that whichever XML parser we use supports XInclude (possibly using a third-party library) as well as the usual JAXP support.

Conclusion

Areas we can collaborate on

Assuming we agree on this general architecture, the steps we could collaborate on are:

1. Identify any areas in this document that are just plain wrong
2. Choose package names
3. Choose a directory structure
4. Identify and define key interfaces
5. Provide implementations to get a minimal search and gene page up
6. Prioritise, and then do, other implementations and pages

I don't think points 1-5 will take that long (“CPU time”!). Up to there perhaps we should just concentrate on the gusweb site ie the generic one. Then re-examine the priorities.

Appendix A - JavaServer Pages

JavaServer Pages (JSPs) are best viewed as special web pages. They come in two flavours: a traditional HTML flavour and a strict XML version.We'll only consider the former. The main contents of a JSP page are detailed below:

HTML tags

The other features below are interspersed in standard HTML pages

JSP custom tags

The cleanest way to extend a JSP is via custom tag libraries. These are pieces of Java code that have a defined life-cycle (like applets and servlets). We need to provide an implementation and a definition file for each library.

In the demo there's a couple of examples. This is a slightly edited copy:

```
<f:table id="resultsTable">
<c:forEach items="${results.rowsByIndex}" begin="${start}"
           end="${start+numResultRows-1}" var="gene" varStatus="s">
  <f:tr>
    <f:td format="int">
      <c:out value="${s.index +1}" />
    </f:td>
    <f:td format="nickname">
      <c:out value="${gene[1]}" default=" " escapeXml="false" />
    </f:td>
    <f:td>
      <c:out value="${gene[0]}" default=" " escapeXml="false" />
    </f:td>
    <f:td format="href" extra1="${gene[1]}">
      <c:out value="${gene[2]}" default=" " escapeXml="false" />
    </f:td>
    <f:td>
      <c:out value="${gene[3]}" default=" " escapeXml="false" />
    </f:td>
    <f:td>
      <c:out value="${gene[4]}" default=" " escapeXml="false" />
    </f:td>
  </f:tr>
</c:forEach>
</f:table>
```

There are two tag libraries in use here. Note that this excerpt just shows custom tags but they are usually freely interspersed with HTML tags. The Java standard core tag library (JSTL), in the 'c' namespace, is used to provide simple logic structures, eg the loop over the result set. The 'f' namespace is being used for a custom formatting tag library. This example shows the table tags. The <f:tr> is like a standard <tr> tag but it understands that alternating rows should have a different background colour. The colours (and exact colouring row pattern) are given in a property file. The <f:td> is more interesting. It takes a format attribute. This is analagous to the CellFormatter in the current system.

There isn't too much programming overhead. The relevant section of the definition file is simply:

```
    <tag>
         <name>tr</name>
         <tagclass>org.genedb.gusweb.taglib.TrTag</tagclass>
         <bodycontent>JSP</bodycontent>
         <info>Defines a &lt;tr&gt; tag that knows how to colour itself</info>
```

```
        <attribute>
               <name>header</name>
        </attribute>
    </tag>
```

And the implementation code (without comments/imports etc) is just:

```
public class TrTag extends TagSupport {

  private boolean header = false;

  public void setHeader(String set) {
    if ("true".equalsIgnoreCase(set)) {
      header = true;
    }
  }


  public int doStartTag() throws JspException {
    TableTag parent = (TableTag) findAncestorWithClass(this, TableTag.class);
    JspWriter out = pageContext.getOut();
    String col = null;

    if (header) {
      col = parent.getHeaderColor();
    } else {
      col = parent.getRowColor();
    }

    try {
      out.print("<tr bgcolor=\"");
      out.print(col);
      out.println("\">");
    }
    catch (IOException exp) {
      throw new JspTagException("I/O exception "+exp.getMessage());
    }

    return Tag.EVAL_PAGE;
  }

  public int doEndTag() throws JspException {
    JspWriter out = pageContext.getOut();
    try {
      out.println("</tr>");
    }
    catch (IOException exp) {
      throw new JspTagException("I/O exception "+exp.getMessage());
    }
    return Tag.EVAL_PAGE;
  }
}
```

I would imagine we could end up with at least 3 different tag libraries.

1. Shared custom format tags. The row colouring example above and the context-sensitive navigation bar are examples of things that all the sites might (or do) use which could be shared and reused with just a different property file.
2. Shared custom logic tags. Some of the data structures coming from the backend could be quite complex. Custom tags for displaying these could be useful. As another example our curators are keen to have the classic Mac outline triangle for hiding/displaying sections in a long page. This could also be usefully shared if any of the other sites are interested.

3. Site specific custom format tags

Directives

Directives provide information about a JSP page to the software that processes the page. Directives do not produce visible output, but rather provide instructions and settings that determine how a JSP page is processed.

Include Directive

The include directive allows you to use one file in several different JSP pages. The key difference between this and jsp:include is that this is resolved at compile time rather than run-time. Hence it's useful for static imports to avoid extra processing overhead.

Taglib Directive

The taglib directive allows you to define a tag library and a prefix that can be used to reference the custom tags.

eg `<%@ taglib uri="http://www.genedb.org/taglib/format" prefix="format" %>`

Page Directive

The page directive allows you to specify information about the configuration of a JSP page. There are a number of options available so they're listed in a table. The underlined are literal text:

Name	Values	Default Value	Description
language	language	java	Scripting language used in the page. Java is required to be supported, particular engines may support others
extends	package.class		Specify that the servlet that this page is eventually compiled into extends a given class.
import	package.class| package.*, ...		Import other classes that are referenced in this page. Like a normal Java program java.lang.* is automatically imported but in addition so are javax.servlet.*, javax.servlet.jsp.*, javax.servlet.http.*
isThreadSafe	true|false	true	Can the page be accessed by multiple threads?
session	true|false	true	Is the page part of the default session handling? If true and a session isn't already in progress a new one will be created before the page is run (loosely speaking)
info	text		A description String that is returned by Servlet.getServletInfo()
buffer	none|sizekb	8kb	The output of the servlet is normally buffered before being sent back to the client. This controls how big the buffer should be
autoFlush	true|false	true	If true, the buffer is flushed ie sent to the browser when it's full. If false the buffer must be flushed (by the end of the page or an explicit flush) before it fills up or an exception will occur.
errorPage	relativeURL		Page to redirect to in case of an error
isErrorPage	true|false	false	Is this page an error page? From the page designer's point of view the main difference is whether the implicit exception object is available.
contentType	MimeType [; charset= characterSet]	text/html ; charset =iso-8859-1	Specify what type of page is returned and its characater set. The default is an HTML page but JSPs are limited to that. You might want to return eg a sequence as text/plain or a diagram as SVG. You can in theory also return binary streams although because of whitespace handling I'd probably just use a servlet for that.

JSP expression

An expression is a scripting element that allows you to generate output on a JSP page. It's delimited by `<%=` and `%>`. in the middle is a Java expression for example a method call or the name of a variable. If the resulting object isn't a String the toString method is called on it. This output is included into the page at that point. So if pageTitle is a variable declared earlier in the page you can use eg `<title><%= pageTitle %></title>` and `<h1><%= pageTitle %></h1>`.

jsp: elements*

jsp:include

Include the contents of another web resource at this point. This content is evaluated dynamically.

jsp:forward

Forwards control to a new web resource. Note you can only specify a relative URL.

jsp:param

In conjuction with either of the above two, you can add key/value parameters to be passed to the web resource.

jsp:useBean

The `<jsp:useBean>` element locates or instantiates a JavaBeans component. `<jsp:useBean>` first attempts to locate an instance of the bean. If the bean does not exist, it is instantiated from a class or serialized template.

jsp:getProperty, jsp:setProperty

Having identified a bean using useBean, you can use these two tags to access a bean's properties (using its accessor methods)

jsp:plugin

Causes the execution of an applet or bean. The applet or bean executes in the specified plugin.

If the plugin is not available, the client displays a dialog to initiate the download of the plugin software.

JSP comment

Comments can be put into the JSP page between `<%--` and `--%>`. These aren't sent back to the browser, and are just to aid the JSP page author. (Obviously HTML comments can also be used which are sent back as usual)

JSP custom tags

Did I mention the main part of the page should be HTML and custom tags?

Code Fragment (Scriplet)

You can include larger blocks of code between `<%` and `%>`. Before custom tags this was the main way of adding functionality to a page. As it mixes Java and HTML it's probably best avoided in production code, but can still be very useful in development.

Declaration

A declaration is a scripting element that allows you to define variables and methods that will be used throughout a JSP page. You must define variables and methods in a JSP page before you can use the variables and methods in the page. Some sources recommend defining all variables in a declartion

but unless they're truly global it seems more sensible, IMO, to restrict their scope to the relevant scriptlet. To create a declaration, you place the code for the declaration between `<%!` and `%>`

The same caveats about mixing logic and presentation that applied to scriptlets also applies here.

From the container's point of view JSP pages are code. Each page can be converted into pure Java code. The plain text is wrapped in print statements, expressions are used verbatim. Directives map either to Java level constructs like import statements or to direct method calls to the container. These pages also implement the Servlet interface and after compilation are treated exactly the same by the container. (You can put code in the JSP page that is called at specific points in the servlet lifecycle – if you really want to!). The page compilation can occur either when the page is first requested or can be precompiled eg as part of the deployment process.

The scripting language used doesn't have to be Java. That's the only language support mandated by the specification but for example Tomcat also supports Python and Tcl. All our examples use Java.