JSP

Contents

The webpages in NetarchiveSuite are written using JSP (Java Server Pages) with Apache Jakarta I18N Taglib for internationalization. To support a unified look across pages from different modules, we have divided the pages into SiteSections as described in the next section. Any processing of requests happens in Java code before the web page is displayed, such that update errors can be handled with a uniform error page. Internationalization is primarily done with the taglib tags <fmt:message>, <fmt:date> etc.

The main feature of JSP is that ordinary Java (not JavaScript) can be used at server-side to generate HTML. The special tags <%...%> indicate a piece of Java code to run, while the tags <%=...> indicates a Java expression to run whose value will be inserted (as is, see escape mechanisms below) in the HTML. While it is possible to output to HTML from Java code using **out.print()**, it is discouraged as it a) is confusing to read, and b) does not allow for using taglibs for internationalization.

We use a number of standard methods defined in **dk.netarkivet.common.webinterface.HTMLUtils**. Of particular note are the following methods:

generateHeader():: This method takes a **PageContext** and generates the full header part of the HTML page, including the starting <body> tag. It should always be used to create the header, as it also creates the menu and language selection links. After this method has been called, redirection or forwarding is no longer possible, so any processing that can cause fatal errors must be done before calling **generateHeader()**. The title of the page is taken from the **SiteSection** information based on the URL used in the request.

generateFooter():: This closes the HTML page and should be called as the last thing on any JSP page.

setUTF8():: This method must be called at the very start of the JSP page to ensure that reading from the request is handled in UTF-8.

encode():: This encodes any character that is not legal to have in a URL. It should be used whenever an unknown string (or a string with known illegal characters) is made part of a URL. Note that it is not idempotent; calling it twice on a string is likely to create a mess.

escapeHTML():: This escapes any character that has special meaning in HTML (such as < or &). It should be used any time an unknown string (or a string with known special characters) is being put into HTML. Note that it is **not** idempotent: If you escape something twice, you get a horrible-looking mess.

encodeAndEscape():: This method combines **encode()** and **escapeHTML()** in one, which is useful when you're putting unknown strings directly into URLs in HTML.

The SiteSection system

Each part of the web site (as identified by the top-level menu items on the left side) is defined by one subclass of the SiteSection class. These sections are loaded through the <siteSection> settings, each of which connect one SiteSection class with its WAR file and the path it will appear under in the URL.

Each SiteSection subclass defines the name used in the left-hand menu, the prefix of all its pages, the number of pages visible in the left-hand menu when within this section, a suffix and title for each page in the section (including hidden pages), the directory that the section should be deployed under, and a resource bundle name for translations. Furthermore, the SiteSections have hooks for code that needs to be run during deployment and shutdown. If you want to add a new page to the section, you will only need to add a new line to the list of pages with a unique (within the SiteSection) suffix and a key for the page title, plus a default translation in the corresponding Translation.properties file. If you want it to appear in the left-hand menu, update the number of visible pages to n+1 and put your new pages as one of the first n+1 lines.

If the page ends with a "/" like in "history/" it means that the page is an external page not included in the war-file and thus the page is not prefixed and suffixed as the rest (ie. the page is "/History/history/" . Compare with the "alljobs" - entry below which ends up as the link /History/Harveststatus-alljobs.jsp".

This is an example of what a simple SiteSection can look like. Note that only the first three pages from the list have entries in the left-hand menu. This class does not do any special initialisation and shutdown.

```
public HistorySiteSection() {
        super("sitesection;history", "Harveststatus", 3,
              new String[][]{
                   {"alljobs", "pagetitle;all.jobs"},
                  {"running", "pagetitle;all.jobs.running"},
                    {"history/", "pagetitle;h3.remote.access"}, // refer
to webpage outside of webpages directory
                    {"running-jobdetails",
"pagetitle;running.job.details"},
                    {"perhd",
"pagetitle;all.jobs.per.harvestdefinition"},
                    {"perharvestrun",
"pagetitle;all.jobs.per.harvestrun"},
                    {"jobdetails", "pagetitle;details.for.job"},
                    {"perdomain", "pagetitle;all.jobs.per.domain"},
                    {"seeds", "pagetitle;seeds.for.harvestdefinition"}
              }, "History",
                 dk.netarkivet.harvester.Constants.TRANSLATIONS_BUNDLE);
    }

    public void intitialize() {}
    public void close() {}
```

Processing updates

Some JSP sites cause updates when posted with specific parameters. Such parameters should always be specified in the beginning of the JSP file. All updates of underlying file systems, databases etc should happen before **generateHeader()** is called, so processing errors can be properly redirected. The preferred way to process updates is to create a method processRequest() in a class corresponding to the web page, but under the **webinterface** package of the corresponding module. This method should take the **pageContext** and **I18N** parameters from the JSP page; together they contain all the information needed from there.

In case of processing errors, the processing method should call **HTMLUtils.forwardToErrorPage()** and then throw a **ForwardedToErrorPage** exception. The JSP code should always enclose the **processRequest()** call in a try-catch block and return immediately if **ForwardedToErrorPage** is thrown, like in the following code-fragment:

```
try {
        HTMLUtils.forwardOnEmptyParameter(pageContext,
           Constants.DOMAIN_PARAM);
        DomainConfigurationDefinition.processRequest(
           pageContext, I18N);
    } catch (ForwardedToErrorPage e) {
        return;
    }
```

This mechanism should be used for "expected" errors, mainly illegal parameters. Errors of the "this can never happen" nature should just cause normal exceptions. Like in other code, the **processRequest()** method should check its parameters, but it should also check the parameters posted in the request to check that they conform to the requirements. Some methods for that purpose can be found in **HTMLUtils**.

I18n

We use the Apache I18n taglib for most internationalization on the web pages. This means that instead of writing different versions of a web

page for different languages, we replace all natural language parts of the page with special formatting instructions. These are then used to look up translations to the language in effect in translation resource bundles.

Normal strings can be handled with the <fmt:message/> tag. If variable parameters are introduced, such as object names or domain names, they can be passed as parameters using <fmt:message key="translation.key"><fmt:param value="<%myVal>"/></fmt:message>. Note that while the message retrieved for the key gets any HTML-specific characters escaped, the values do not and should be manually escaped. It is possible if necessary to pass HTML as parameters.

Dates should in general be entered using <fmt:formatDate type="both">, though a few places use a more explicit handling of formats. This lets the date be expressed in the native language's favorite style.

Integers and longs are handled in Java properties files with {0, number, integer} or {0, number, long} as described in MessageFormat. In JSP files we convert integer/long to strings with method HTMLUtils.localiseLong(long, PageContext) or method HTMLUtil.localiseLong(long, Locale).

Note the boilerplate code at the start of every page that defines output encoding, taglib usage, translation bundle, and a general-purpose I18N object. It is important that the translation bundles from the **Constants** class for the module you're in is used, or incomprehensible errors will occur.

```
pageEncoding="UTF-8"
%><%@taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt"
%><fmt:setLocale value="<%=HTMLUtils.getLocale(request)%>" scope="page"
/><fmt:setBundle scope="page"
basename="<%=dk.netarkivet.archive.Constants.TRANSLATIONS_BUNDLE%>"/><%
    private static final I18n I18N
            = new
I18n(dk.netarkivet.archive.Constants.TRANSLATIONS_BUNDLE);
%><%
```