Implementing Hamlets

Further refining this Web development framework

Ren<eacute /> Pawlitzek (rpa@zurich.ibm.com)
Research and Development Engineer
IBM

Skill Level: Intermediate

Date: 07 Feb 2006
(Updated 03 Jul 2007)

The Hamlet framework was developed to extend Java™ servlets and enforce the separation of content from presentation. In this article, you'll find an additional way to provide dynamic content as René Pawlitzek advances the framework further and refines use of the template engine.

Hamlets provide an easily used and easily understood framework for developing Web-based applications. The framework not only supports, but also enforces, the complete separation of content and presentation. Its simple and elegant design does not hide the familiar underlying servlet infrastructure.

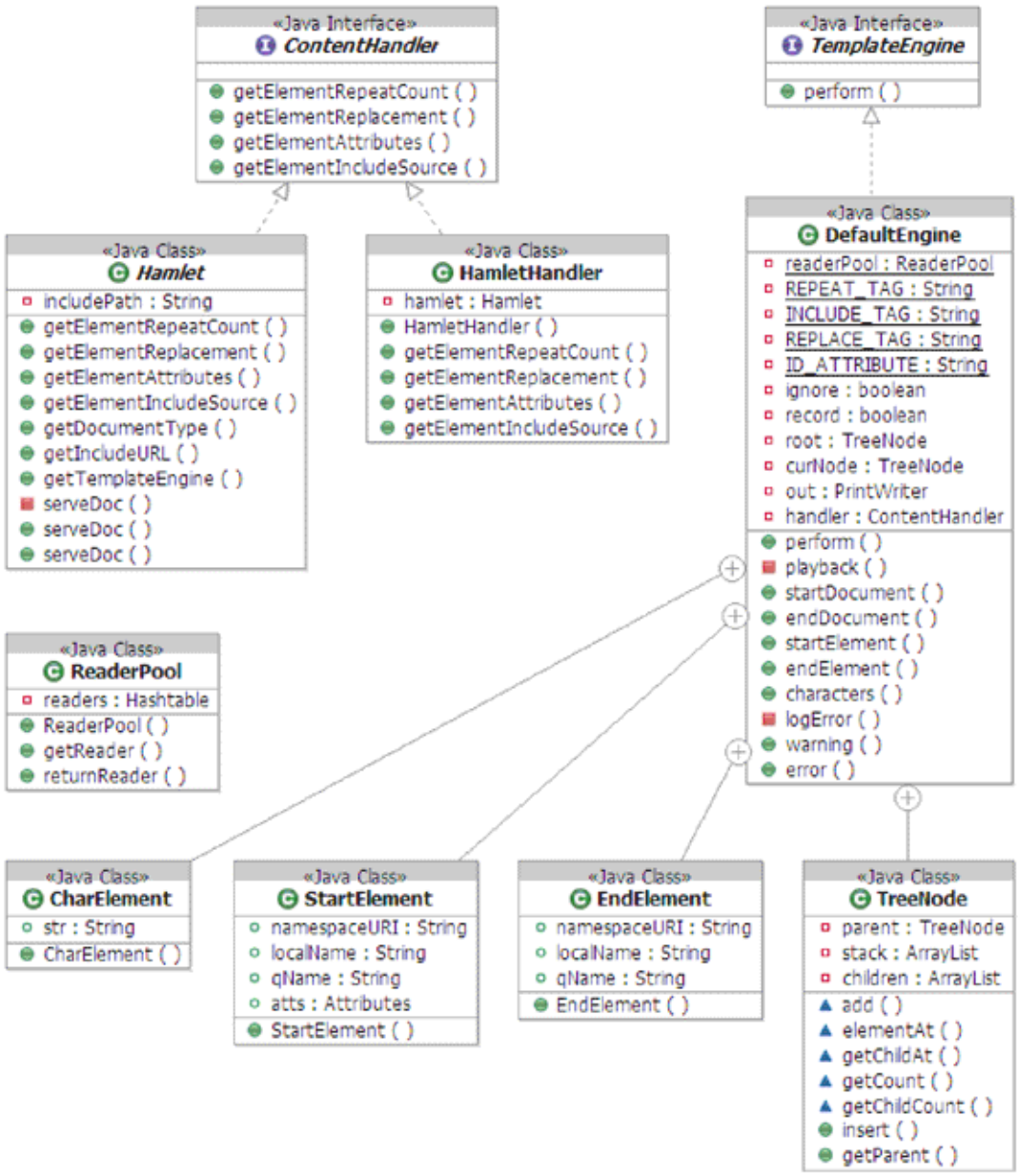
The framework provides a servlet extension called a *Hamlet.* A Hamlet uses the Simple API for XML (SAX) to read template files containing presentation. (These template files must contain content that can be parsed by SAX, such as strict HTML, XHTML, or XML.) While a template file is being read, the Hamlet uses a small set of callback functions to dynamically add content to those places in the template that are marked with special tags and IDs.

The project name *Hamlets* used in this publication refers to an internal project for the development of a servlet-based framework for separation of content and presentation in Web-based applications. Before you read further in this article, you should read my previous articles in this series on IBM developerWorks: Introducing Hamlets, and, optionally, my developerWorks tutorial, Programming Hamlets. (See Resources for links to both.)

This article illustrates various aspects of implementing Hamlets. A basic Hamlet implementation (Version 1.0) using just two public Java classes with less than 500 lines of Java code was shown in the "Programming Hamlets" tutorial. The code included here (Version 1.1) contains two Java interfaces, four public Java classes, and a total of 695 lines of Java code; this implementation is more advanced and

better structured (see the UML diagram in Figure 1), and is explained in detail in this article. Furthermore, in this article you'll learn about a new way to program Hamlets using `HamletHandler`s. You can download wa-hamlets3-code.zip, an archive containing the complete sample code from this article.

Figure 1. UML diagram for Hamlets 1.1



The TemplateEngine and ContentHandler interfaces and the DefaultEngine class

At the core of the Hamlet framework is a *template engine* -- a class that implements the `TemplateEngine` interface, shown in Listing 1.

Listing 1. The TemplateEngine interface defines a template engine's functionality

```
public interface TemplateEngine {

  public void perform (InputStream in, ContentHandler handler, PrintWriter out)
    throws Exception;

} // TemplateEngine
```

A template engine provides a public method, `perform()`, that takes a template and parses it. While the template is parsed, a number of callback functions are invoked to generate the dynamic content that the engine fills into the template. The input (the template) is given through an `InputStream` and the output (the filled template) is collected with the help of a `PrintWriter`. A handler object provides the callback functions and implements the Hamlet `ContentHandler` interface shown in Listing 2.

Listing 2. The ContentHandler interface defines the callback function signatures

```
public interface ContentHandler {

  public int getElementRepeatCount (String id, String name, Attributes atts)
    throws Exception;

  public String getElementReplacement (String id, String name, Attributes atts)
    throws Exception;

  public Attributes getElementAttributes (String id, String name, Attributes atts)
    throws Exception;

  public InputStream getElementIncludeSource (String id, String name, Attributes atts)
    throws Exception;

} // ContentHandler
```

The `DefaultEngine` class offers a standard implementation of the `TemplateEngine` interface and uses SAX for template parsing. Its `perform()` method is shown in Listing 3.

Listing 3. The DefaultEngine class provides a standard implementation of the TemplateEngine interface

```
public class DefaultEngine extends DefaultHandler implements TemplateEngine {

  ...

  public void perform (InputStream in, ContentHandler handler, PrintWriter out)
    throws Exception {

    XMLReader reader = null;
    try {
      this.out = out;
      this.handler = handler;
      reader = readerPool.getReader ();
      InputSource inputSource = new InputSource (in);
      reader.setErrorHandler (this);
      reader.setContentHandler (this);
      reader.parse (inputSource);
    } finally {
      if (reader != null)
```

```
        readerPool.returnReader (reader);
    } // try

  } // perform

  ...

} // DefaultEngine
```

The code in Listing 3 obtains a SAX reader from a pool of readers, wraps the input parameter `in` with `InputSource`, sets the SAX error and content handlers, and starts to parse the input source (the template). It returns the SAX reader to the pool of readers when the parsing is finished. SAX reader buffering is done for performance reasons because creating a new SAX reader for each request would take too much time.

During the parsing of a template, the engine's SAX content handler methods (`startDocument()`, `endDocument()`, `startElement()`, `endElement()`, and `characters()`) are called. They are implemented as shown in Listing 4.

Listing 4. The DefaultEngine class implements the SAX content handler methods

```
public class DefaultEngine extends DefaultHandler implements TemplateEngine {

  ...

  public void startDocument () throws SAXException {
    root = new TreeNode ();
    curNode = root;
  } // startDocument


  public void endDocument () throws SAXException {
    curNode = null;
    root = null;
  } // endDocument


  public void startElement (String namespaceURI, String localName, String qName,
    Attributes atts) throws SAXException {

    try {
      // category.debug ("local name: " + localName + ", qname: " + qName);
      if (REPEAT_TAG.equals (localName)) {
        record = true;
        StartElement elem = new StartElement (namespaceURI, localName, qName, atts);
        curNode.add (elem);
        TreeNode newNode = new TreeNode ();
        curNode.insert (newNode, 0);
        curNode = newNode;
      } else if (REPLACE_TAG.equals (localName)) {
        if (record) {
          StartElement elem = new StartElement (namespaceURI, localName, qName, atts);
          curNode.add (elem);
        } else {
          ignore = true;
          String id = atts.getValue (ID_ATTRIBUTE);
          if (id != null)
            out.print (handler.getElementReplacement (id, localName, atts));
        } // if
```

```
    } else if (INCLUDE_TAG.equals (localName)) {
      if (record) {
        StartElement elem = new StartElement (namespaceURI, localName, qName, atts);
        curNode.add (elem);
      } else {
        ignore = true;
        String id = atts.getValue (ID_ATTRIBUTE);
        if (id != null)
          atts = handler.getElementAttributes (id, localName, atts);
        InputStream in = handler.getElementIncludeSource (id, localName, atts);
        if (in != null) {
          String line;
          BufferedReader r = new BufferedReader (new InputStreamReader (in));
          while ((line = r.readLine ()) != null)
            out.println (line);
          r.close ();
        } // if
      } // if
    } else {
      if (record) {
        StartElement elem = new StartElement (namespaceURI, localName, qName, atts);
        curNode.add (elem);
      } else {
        String id = atts.getValue (ID_ATTRIBUTE);
        if (id != null)
          atts = handler.getElementAttributes (id, localName, atts);
        out.print ("<" + localName);
        for (int i = 0; i < atts.getLength (); i++) {
          out.print (" ");
          out.print (atts.getLocalName (i));
          out.print ("=\"");
          out.print (atts.getValue (i));
          out.print ("\"");
        } // for
        out.print (">");
      } // if
    } // if
  } catch (Exception e) {
    category.debug (e.getMessage (), e);
    throw new SAXException (e);
  } // try

} // startElement


public void endElement (String namespaceURI, String localName, String qName)
  throws SAXException {

  try {
    if (REPEAT_TAG.equals (localName)) {
      curNode = (TreeNode) curNode.getParent ();
      EndElement elem = new EndElement (namespaceURI, localName, qName);
      curNode.add (elem);
      if (curNode.equals (root)) {
        record = false;
        playback (root);
        root = new TreeNode ();
        curNode = root;
      } // if
    } else if (REPLACE_TAG.equals (localName) || INCLUDE_TAG.equals (localName)) {
      if (record) {
        EndElement elem = new EndElement (namespaceURI, localName, qName);
        curNode.add (elem);
      } else {
        ignore = false;
      } // if
    } else {
```

```
        if (record) {
          EndElement elem = new EndElement (namespaceURI, localName, qName);
          curNode.add (elem);
        } else {
          out.print ("</" + localName + ">");
        } // if
      } // if
    } catch (Exception e) {
      category.debug (e.getMessage (), e);
      throw new SAXException (e);
    } // try

  } // endElement


  public void characters (char[] ch, int start, int length) throws SAXException {
    String str = new String (ch, start, length);
    if (record) {
      CharElement elem = new CharElement (str);
      curNode.add (elem);
    } else {
      if (!ignore)
        out.print (str);
    } // if
  } // characters

  ...

} // DefaultEngine
```

The `startDocument()` method is called before the template is actually parsed. It initializes two instance variables (`root` and `curNode`) used for recording purposes. After the parser finishes, the `endDocument()` method sets both instance variables to `null`.

The SAX parser calls the `startElement()` and `endElement()` methods whenever it recognizes starting and ending tags. Special action takes place when it encounters any of the following tags:

- `<REPEAT>`
- `</REPEAT>`
- `<REPLACE>`
- `</REPLACE>`
- `<INCLUDE>`
- `</INCLUDE>`

These tags are embedded into template files to mark places where dynamic content is added during the parsing.

When the SAX reader comes across a `<REPEAT>` tag, the `startElement()` method sets the `record` flag to `true`, as shown in Listing 5, and thus all following content in the template file is recorded with the help of a tree structure and the four inner classes `TreeNode`, `StartElement`, `EndElement`, and `CharElement`. The tree structure is necessary because `<REPEAT>` tags can nest.

Listing 5. The <REPEAT> tag processing in startElement() starts the content recording

```
    if (REPEAT_TAG.equals (localName)) {
      record = true;
      ...
    }
```

To stop the recording, the `endElement()` method sets the `record` flag to `false` when the corresponding `</REPEAT>` tag is encountered, as shown in Listing 6.

Listing 6. The </REPEAT> tag processing in endElement() stops the content recording and invokes the content playback

```
    if (REPEAT_TAG.equals (localName)) {
      ...
      record = false;
      playback (root);
      ...
    }
```

The `endElement()` method also invokes the `playback()` function, shown in Listing 7, which calls the `ContentHandler`'s `getElementRepeatCount()` callback to retrieve the number of repeats, `N`. (The `getElementRepeatCount()` method is also called repeatedly during playback to enable termination by returning 0.) Next, the recorded content (a tree of elements) plays back `N` times. To play back recorded elements, call the `startElement()`, `endElement()`, or `characters()` method; these methods treat recorded content just like they treat non-recorded content. Note that `playback()` is recursive because `<REPEAT>` tags can nest.

Listing 7. The playback() function repeats recorded content N times

```
  private void playback (TreeNode node) throws Exception {
    int childIndex = node.getChildCount () - 1;
    for (int i = 0; i < node.getCount (); i++) {
      Object obj = node.elementAt (i);
      if (obj instanceof StartElement) {
        StartElement elem = (StartElement) obj;
        if (REPEAT_TAG.equals (elem.localName)) {
          int count = 0;
          String id = elem.atts.getValue (ID_ATTRIBUTE);
          if (id != null)
            count = handler.getElementRepeatCount (id, elem.localName, elem.atts);
          for (int j = 0; j < count; j++) {
            playback ((TreeNode) node.getChildAt (childIndex));
            if (handler.getElementRepeatCount (id, elem.localName, elem.atts) == 0)
              break;
          } // if
          childIndex--;
        } else
          startElement (elem.namespaceURI, elem.localName, elem.qName, elem.atts);
      } else if (obj instanceof EndElement) {
        EndElement elem = (EndElement) obj;
        if (!REPEAT_TAG.equals (elem.localName))
          endElement (elem.namespaceURI, elem.localName, elem.qName);
      } else if (obj instanceof CharElement) {
        CharElement elem = (CharElement) obj;
        characters (elem.str.toCharArray (), 0, elem.str.length ());
```

```
    } // if
  } // for
} // playback
```

When the SAX reader encounters a `<REPLACE>` tag and recording is deactivated, the `startElement()` method invokes the `ContentHandler`'s `getElementReplacement()` method if an `ID` attribute exists. The `getElementReplacement()` callback returns dynamically created content that is included in the output with the `print()` method. The `ignore` flag was previously set to `true` to ensure that the placeholder value in the template file (the content between the `<REPLACE>` and `</REPLACE>` tags) is ignored and not included in the output by the `characters()` method.

Listing 8. <REPLACE> tag processing in startElement()

```
    if (REPLACE_TAG.equals (localName)) {
      if (record) {
        ...
      } else {
        ignore = true;
        String id = atts.getValue (ID_ATTRIBUTE);
        if (id != null)
          out.print (handler.getElementReplacement (id, localName, atts));
      } // if
    } ...
```

Similarly, when the SAX reader comes across an `<INCLUDE>` tag and recording is switched off, the `startElement()` method calls the `ContentHandler`'s `getElementIncludeSource()` method to retrieve content that is then included in the output. Prior to this, the `getElementAttributes()` callback is invoked in order to enable a Hamlet to modify the attributes if the `<INCLUDE>` tag contains an `ID` attribute. Furthermore, the `ignore` flag is set to `true` to ignore the placeholder value in the template file (the content between the `<INCLUDE>` and `</INCLUDE>` tags).

Listing 9. <INCLUDE> tag processing in startElement()

```
    if (INCLUDE_TAG.equals (localName)) {
      if (record) {
        ...
      } else {
        ignore = true;
        String id = atts.getValue (ID_ATTRIBUTE);
        if (id != null)
          atts = handler.getElementAttributes (id, localName, atts);
        InputStream in = handler.getElementIncludeSource (id, localName, atts);
        if (in != null) {
          String line;
          BufferedReader r = new BufferedReader (new InputStreamReader (in));
          while ((line = r.readLine ()) != null)
            out.println (line);
          r.close ();
        } // if
      } // if
    } ...
```

The `ignore` flag is set to `false` in the `endElement()` method when the SAX reader comes across a `</REPLACE>` or an `</INCLUDE>` tag, as shown in Listing 10.

Listing 10. </REPEAT> tag and </INCLUDE> tag processing in endElement()

```
      if (REPLACE_TAG.equals (localName) || INCLUDE_TAG.equals (localName)) {
        if (record) {
          ...
        } else {
          ignore = false;
        } // if
      } ...
```

In non-record mode and for all other tags, the `startElement()` method calls `getElementAttributes()` if the tag contains an ID attribute. The `getElementAttributes()` callback can add, remove, and change the attributes of a tag. Next, the tag with its possibly modified attributes is written to the output stream. This process is illustrated in Listing 11.

Listing 11. Process ordinary starting tags in startElement()

```
      } else {
        if (record) {
          ...
        } else {
          String id = atts.getValue (ID_ATTRIBUTE);
          if (id != null)
            atts = handler.getElementAttributes (id, localName, atts);
          out.print ("<" + localName);
          for (int i = 0; i < atts.getLength (); i++) {
            out.print (" ");
            out.print (atts.getLocalName (i));
            out.print ("=\"");
            out.print (atts.getValue (i));
            out.print ("\"");
          } // for
          out.print (">");
        } // if
      } ...
```

The corresponding ending tag is generated in the `endElement()` method with the `print()` command, as shown in Listing 12.

Listing 12. Process ordinary ending tags in endElement()

```
      } else {
        if (record) {
          ...
        } else {
          out.print ("</" + localName + ">");
        } // if
      } ...
```

The Hamlet class

The abstract class `Hamlet`, shown in Listing 13, is an `HttpServlet` extension and implements the Hamlet `ContentHandler` interface:

Listing 13. The Hamlet class implements the ContentHandler interface

```
public abstract class Hamlet extends HttpServlet implements ContentHandler {


  ...
```

```
  private  String  includePath;


  public int getElementRepeatCount (String id, String name, Attributes atts)
    throws Exception {
    return 0;
  } // getElementRepeatCount


  public String getElementReplacement (String id, String name, Attributes atts)
    throws Exception {
    return "";
  } // getElementReplacement


  public Attributes getElementAttributes (String id, String name, Attributes atts)
    throws Exception {
    return atts;
  } // getElementAttributes


  public InputStream getElementIncludeSource (String id, String name, Attributes atts)
    throws Exception {
    URL url = new URL (getIncludeURL (atts.getValue ("SRC")));
    return url.openStream ();
  } // getElementIncludeSource


  public String getDocumentType () {
    return "text/html";
  } // getDocumentType


  public String getIncludeURL (String fileName) {
    return includePath + fileName;
  } // getIncludeURL


  public TemplateEngine getTemplateEngine () {
    return new DefaultEngine ();
  } // getTemplateEngine


  private void serveDoc (PrintWriter out, String template, ContentHandler handler)
    throws Exception {
    TemplateEngine engine = getTemplateEngine ();
    InputStream in = getServletContext().getResourceAsStream (template);
    category.debug ("Parsing '" + template + "' ...");
    long t1 = System.currentTimeMillis ();
    engine.perform (in, handler, out);
    long t2 = System.currentTimeMillis ();
    category.debug ("Parsed '" + template + "' in " + (t2 - t1) + " ms.");
  } // serveDoc


  public void serveDoc (HttpServletRequest req, HttpServletResponse res,
    String template, ContentHandler handler) throws Exception {
    includePath = "http://localhost:" + req.getServerPort () + "/" +
      req.getContextPath () + "/";
    PrintWriter out = res.getWriter ();
    res.setContentType (getDocumentType ());
    serveDoc (out, template, handler);
  } // serveDoc


  public void serveDoc (HttpServletRequest req, HttpServletResponse res,
    String template) throws Exception {
```

```
    serveDoc (req, res, template, this);
  } // serveDoc


} // Hamlet
```

The `Hamlet` class provides three `serveDoc()` methods. `serveDoc(PrintWriter out, String template, ContentHandler handler)` is private and is the most interesting of the three. It calls `getTemplateEngine()` to obtain a default template engine, retrieves the template with a `getResourceAsStream()` call, and invokes the engine's `perform()` method. The other two `serveDoc()` methods are public and call `serveDoc(PrintWriter out, String template, ContentHandler handler)` either directly or indirectly.

Furthermore, the abstract class `Hamlet` implements the Hamlet `ContentHandler` interface to provide a default implementation for the callback methods `getElementReplacement()`, `getElementRepeatCount()`, `getElementAttributes()`, and `getElementIncludeSource()`. By deriving a class from the `Hamlet` class, one can provide a specific implementation for the callback functions to fill a particular template.

The `Logout1` class in Listing 14 is an extension of `Hamlet`. It overwrites the `getElementReplacement()` method to fill the template LogoutTemplate.html with the user ID.

Listing 14. The Logout1 class provides the dynamic content for LogoutTemplate.html

```
public class Logout1 extends Hamlet {


  // log4j
  private static Category category = Category.getInstance (Logout1.class.getName ());


  private  String  userID = null;


  public void init () throws ServletException {
    // get properties
    ContextProperties props = ContextProperties.getProperties (this);
    // configure logging
    Utilities.configLog (props);
    category.debug ("init");
  } // init


  public synchronized void doGet (HttpServletRequest req, HttpServletResponse res)
    throws ServletException {

    try {
      category.debug ("doGet");
      HttpSession session = req.getSession (false);
      if (session != null) {
        WebApp webApp = (WebApp) session.getAttribute (WebApp.ID);
        userID = webApp.getUserID ();
        // invalidate session
        session.invalidate ();
```

```
        // serve document
        serveDoc (req, res, "LogoutTemplate.html");
      } else {
        // redirect
        res.sendRedirect (res.encodeURL (req.getContextPath () +
          "/StaticPage.html?Page=LoginHere.html"));
      } // if
    } catch (Exception e) {
      category.debug ("", e);
      throw new ServletException (e);
    } // try

  } // doGet


  public String getElementReplacement (String id, String name, Attributes atts)
    throws Exception {

    if (id.equals ("UserID")) {
      return (userID == null) ? "" : userID;
    } // if
    return "?";

  } // getElementReplacement


} // Logout1
```

The `getElementReplacement()` callback is invoked during the parsing of LogoutTemplate.html by the template engine. The parsing itself is initiated by the `serveDoc(req, res, "LogoutTemplate.html")` method. `Logout1` inherits `serveDoc()` from the `Hamlet` class. The content (the user ID) that `getElementReplacement()` supplies to the template engine is retrieved in the `doGet()` method and stored in the `userID` instance variable of the Hamlet.

A servlet container provides by default only a single instance of a servlet/Hamlet, and therefore the instance variable `userID` is shared among the multiple threads that execute the `doGet()` method for multiple requests. Consequently, `doGet()` must be synchronized to guarantee correctness. Alternatively, however, the `Logout1` Hamlet can implement the empty `SingleThreadModel` interface, as shown in Listing 15. In this case, the servlet container will ensure that no two threads will execute concurrently in the `doGet()` method. Any servlet/Hamlet implementing the `SingleThreadModel` interface can be considered thread safe and is not required to synchronize access to its instance variables (`userID`, in this case). In other words, the `doGet()` method does not need to be synchronized in such cases.

Listing 15. The Logout1 class implements the SingleThreadModel interface to be thread safe

```
public class Logout1 extends Hamlet implements SingleThreadModel {


  // log4j
  private static Category category = Category.getInstance (Logout1.class.getName ());


  private  String  userID = null;
```

```
  public void init () throws ServletException {
    // get properties
    ContextProperties props = ContextProperties.getProperties (this);
    // configure logging
    Utilities.configLog (props);
    category.debug ("init");
  } // init


  public void doGet (HttpServletRequest req, HttpServletResponse res)
    throws ServletException {

    try {
      category.debug ("doGet");
      HttpSession session = req.getSession (false);
      if (session != null) {
        WebApp webApp = (WebApp) session.getAttribute (WebApp.ID);
        userID = webApp.getUserID ();
        // invalidate session
        session.invalidate ();
        // serve document
        serveDoc (req, res, "LogoutTemplate.html");
      } else {
        // redirect
        res.sendRedirect (res.encodeURL (req.getContextPath () +
          "/StaticPage.html?Page=LoginHere.html"));
      } // if
    } catch (Exception e) {
      category.debug ("", e);
      throw new ServletException (e);
    } // try

  } // doGet


  public String getElementReplacement (String id, String name, Attributes atts)
    throws Exception {

    if (id.equals ("UserID")) {
      return (userID == null) ? "" : userID;
    } // if
    return "?";

  } // getElementReplacement


} // Logout1
```

The synchronization of `doGet()` illustrated in Listing 14 disables the concurrent execution of multiple requests. In most cases, this is not an issue if the data for the template filling can be obtained quickly. Synchronize part of the `doGet()` method (as in Listing 16) if data retrieval takes too much time. Template parsing (invoking `serveDoc()`) is very fast in general.

Listing 16. Fine-grained synchronization of doGet() enables concurrent execution

```
  public void doGet (HttpServletRequest req, HttpServletResponse res)
    throws ServletException {

    try {
      category.debug ("doGet");
      HttpSession session = req.getSession (false);
```

```
    if (session != null) {
      WebApp webApp = (WebApp) session.getAttribute (WebApp.ID);
      // obtain user ID, takes some time
      String aUserID = webApp.getUserID ();
      // invalidate session
      session.invalidate ();
      // serve document
      synchronized (this) {
        userID = aUserID;
        serveDoc (req, res, "LogoutTemplate.html");
      } // synchronized
    } else {
      // redirect
      res.sendRedirect (res.encodeURL (req.getContextPath () +
        "/StaticPage.html?Page=LoginHere.html"));
    } // if
  } catch (Exception e) {
    category.debug ("", e);
    throw new ServletException (e);
  } // try

} // doGet
```

Introducing HamletHandler

Using Hamlet instance variables for temporary storage and synchronizing the `doGet()` method or implementing the `SingleThreadModel` interface is an easy way to reduce complexity when writing simple Web-based applications. Hamlets v1.1 has a new, alternative way to write a Hamlet, as shown in Listing 17.

Listing 17. The Logout class uses a HamletHandler extension to provide the dynamic content

```
public class Logout extends Hamlet {


  // log4j
  private static Category category = Category.getInstance (Logout.class.getName ());


  private class LogoutHandler extends HamletHandler {

    private  String  userID = null;

    public LogoutHandler (Hamlet hamlet, String userID) {
      super (hamlet);
      this.userID = userID;
    } // LogoutHandler

    public String getElementReplacement (String id, String name, Attributes atts)
      throws Exception {
      if (id.equals ("UserID")) {
        return (userID == null) ? "" : userID;
      } // if
      return "?";
    } // getElementReplacement

  } // LogoutHandler



  public void init () throws ServletException {
    // get properties
    ContextProperties props = ContextProperties.getProperties (this);
```

```
    // configure logging
    Utilities.configLog (props);
    category.debug ("init");
  } // init



  public void doGet (HttpServletRequest req, HttpServletResponse res)
    throws ServletException {

    try {
      category.debug ("doGet");
      HttpSession session = req.getSession (false);
      if (session != null) {
        WebApp webApp = (WebApp) session.getAttribute (WebApp.ID);
        String userID = webApp.getUserID ();
        // invalidate session
        session.invalidate ();
        // serve document
        HamletHandler handler = new LogoutHandler (this, userID);
        serveDoc (req, res, "LogoutTemplate.html", handler);
      } else {
        // redirect
        res.sendRedirect (res.encodeURL (req.getContextPath () +
          "/StaticPage.html?Page=LoginHere.html"));
      } // if
    } catch (Exception e) {
      category.debug ("", e);
      throw new ServletException (e);
    } // try

  } // doGet


} // Logout
```

The `Logout` class extends `Hamlet`, just as the `Logout1` class does. However, it also contains a private class, `LogoutHandler`, that extends `HamletHandler`. `HamletHandler`, shown in Listing 18, implements the Hamlet `ContentHandler` interface and provides a default implementation (like the `Hamlet` class) for all four callback functions.

Listing 18. The HamletHandler class provides a standard implementation of the ContentHandler interface

```
public class HamletHandler implements ContentHandler {


  private  Hamlet  hamlet;


  public HamletHandler (Hamlet hamlet) {
    this.hamlet = hamlet;

  } // HamletHandler


  public int getElementRepeatCount (String id, String name, Attributes atts)
    throws Exception {
    return 0;
  } // getElementRepeatCount


  public String getElementReplacement (String id, String name, Attributes atts)
    throws Exception {
```

```
    return "";
  } // getElementReplacement


  public Attributes getElementAttributes (String id, String name, Attributes atts)
    throws Exception {
    return atts;
  } // getElementAttributes


  public InputStream getElementIncludeSource (String id, String name, Attributes atts)
    throws Exception {
    URL url = new URL (hamlet.getIncludeURL (atts.getValue ("SRC")));
    return url.openStream ();
  } // getElementIncludeSource

} // HamletHandler
```

The `doGet()` method of the `Logout` class creates an instance of `LogoutHandler` named `handler` and passes it to the `serveDoc()` method. The template engine will call the `LogoutHandler` when dynamic content needs to be filled into the LogoutTemplate.html template. The `LogoutHandler` code returns the user ID to the template engine. The user ID is retrieved in the `doGet()` method but not stored in an instance variable of the Hamlet class. Instead, it is stored in the `userID` instance variable of the `LogoutHandler` class by the constructor. With a `LogoutHandler` created for each request, multiple requests do not need to share a single instance variable of the Hamlet class. The private class `LogoutHandler` adds complexity. On the other hand, it avoids synchronization issues and allows the concurrent execution of multiple requests.

Extending Hamlets

The Hamlets framework is designed with extensibility in mind. You can write your own template engine by implementing the `TemplateEngine` interface. For example, you might want to create an engine that supports namespaces and other requirements. Listing 19 offers an example.

Listing 19. The XMLEngine class provides another implementation of the TemplateEngine interface

```
public class XMLEngine implements TemplateEngine {

  ...

  public void perform (InputStream in, ContentHandler handler, PrintWriter out)
    throws Exception {

    ...

  } // perform

  ...

} // XMLEngine
```

You would instruct the Hamlet to use your custom template engine as shown in Listing 20.

Listing 20. XMLHamlet uses a custom template engine

```
public class XMLHamlet extends Hamlet {


  ...


  private class XMLHamletHandler extends HamletHandler {

    public XMLHamletHandler (Hamlet hamlet) {
      super (hamlet);
    } // XMLHamletHandler

  } // XMLHamletHandler


  public String getDocumentType () {
    return "text/xml";
  } // getDocumentType


  public TemplateEngine getTemplateEngine () {
    return new XMLTemplateEngine ();
  } // getTemplateEngine


  public void doGet (HttpServletRequest req, HttpServletResponse res)
    throws ServletException {

    try {
      // serve document
      HamletHandler handler = new XMLHamletHandler (this);
      serveDoc (req, res, "XMLTemplate.html", handler);
    } catch (Exception e) {
      throw new ServletException (e);
    } // try

  } // doGet


} // XMLHamlet
```

Summary

In comparison with the implementation shown in the earlier Programming Hamlets tutorial (Version 1.0, which uses just two public Java classes with less than 500 lines of code), the more advanced implementation (Version 1.1 containing two Java interfaces, four public Java classes, and 695 lines of Java code) separates the template engine from the Hamlet class. The callback functions are now defined by the `ContentHandler` interface. Furthermore, the new implementation offers an additional way to provide dynamic content to the template engine. With a separate instance of `HamletHandler` for each request, you no longer need to synchronize the `doGet()` methods.

Acknowledgment

I want to thank Yann Duponchel for his suggestions and contributions towards a more advanced implementation of Hamlets.

Downloads

Description	Name	Size	Download method
Complete code sample for this article	wa-hamlets3-code.zip	6KB	HTTP

Information about download methods

Resources

Learn

- "Introducing Hamlets" (developerWorks, March 2005): This article presents the fundamentals of Hamlet programming and explains how to separate content and presentation with just a smidgeon of code.
- "Programming Hamlets" (developerWorks, May 2005): This tutorial illustrates various aspects of Hamlet v1.0 programming as it provides a number of practical Hamlet examples.
- The Java Servlet API White Paper: Get an overview of Java servlets.
- developerWorks Web Architecture zone: Expand your Web authoring skills.

Get products and technologies

- SAX, the Simple API for XML: Check out this widely used API.
- Visit the Enhydra XMLC Project for presentation technology that delivers strict separation of markup and logic in a true object view of dynamic presentations.
- Hamlets on SourceForge: Now that this project is open source, find out how you can contribute.
- Rational® development tools: Download these tools now.

Discuss

- Participate in the discussion forum for this content.
- developerWorks blogs: Get involved in the developerWorks community.

About the author

Ren<eacute /> Pawlitzek

René Pawlitzek is a citizen of Liechtenstein and holds an engineering degree in computer science from the Swiss Federal Institute of Technology (ETH Zürich). René works as a research and development engineer focusing on security information management solutions for the Advanced Operating Environment group (formerly Global Security Analysis Lab (GSAL)) at the IBM Zürich Research Laboratory in Switzerland. Before coming to IBM, he worked in California for Hewlett-Packard, WindRiver Systems, and Borland International.