IBM

developerWorks

Compiling Hamlets

Adding template compilation can improve the performance of this framework

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

Skill Level: Introductory

Date: 20 Jun 2006

René Pawlitzek continues to advance the Hamlets framework, which extends Java servlets and enforces the separation of content and presentation. In this article, he proposes a new refinement: a method of compiling Hamlet templates that can improve application performance.

Today's applications are quite often big, slow, and overly complex. The stack trace in Listing 1, from my first experience developing with JavaServer Faces, helps to illustrate just how convoluted modern code can be.

Listing 1. JSF stack trace

```
java.lang.Exception: Authorization server name not found
  at com.ibm.zurich.gsal.billyboard.libs.IntranetAuthProcessor.configure(Unknown Source)
  at com.ibm.zurich.gsal.billyboard.apps.board.LoginController.<init>(Unknown Source)
  at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
  at sun.reflect.NativeConstructorAccessorImpl.newInstance
        (NativeConstructorAccessorImpl.java:39)
  at sun.reflect.DelegatingConstructorAccessorImpl.newInstance
        (DelegatingConstructorAccessorImpl.java:27)
  at java.lang.reflect.Constructor.newInstance(Constructor.java:274)
  at java.lang.Class.newInstance0(Class.java:308)
  at java.lang.Class.newInstance(Class.java:261)
  at java.beans.Beans.instantiate(Beans.java:204)
  at java.beans.Beans.instantiate(Beans.java:48)
  at com.sun.faces.config.ManagedBeanFactory.newInstance(ManagedBeanFactory.java:203)
  at com.sun.faces.application.ApplicationAssociate.createAndMaybeStoreManagedBeans
        (ApplicationAssociate.java:256)
  at com.sun.faces.el.VariableResolverImpl.resolveVariable(VariableResolverImpl.java:78)
  at com.sun.faces.el.impl.NamedValue.evaluate(NamedValue.java:125)
  at com.sun.faces.el.impl.ComplexValue.evaluate(ComplexValue.java:146)
  at com.sun.faces.el.impl.ExpressionEvaluatorImpl.evaluate
        (ExpressionEvaluatorImpl.java:243)
  at com.sun.faces.el.ValueBindingImpl.getValue(ValueBindingImpl.java:173)
  at com.sun.faces.el.ValueBindingImpl.getValue(ValueBindingImpl.java:154)
  at javax.faces.component.UIOutput.getValue(UIOutput.java:147)
  at com.sun.faces.renderkit.html_basic.HtmlBasicInputRenderer.getValue
       (HtmlBasicInputRenderer.java:82)
  at com.sun.faces.renderkit.html_basic.HtmlBasicRenderer.getCurrentValue
```

© Copyright IBM Corporation 2006

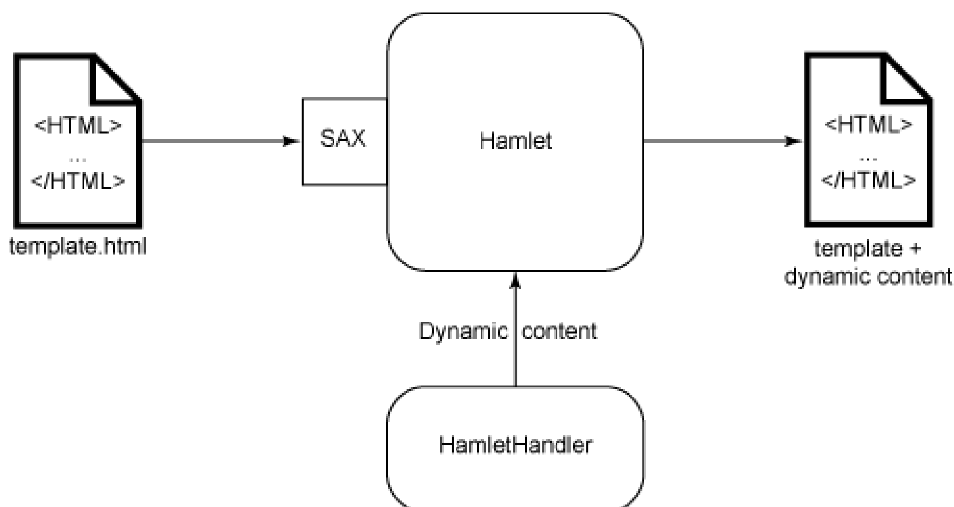
```
     (HtmlBasicRenderer.java:191)
at com.sun.faces.renderkit.html_basic.HtmlBasicRenderer.encodeEnd
     (HtmlBasicRenderer.java:169)
at javax.faces.component.UIComponentBase.encodeEnd(UIComponentBase.java:720)
at com.sun.faces.renderkit.html_basic.HtmlBasicRenderer.encodeRecursive
     (HtmlBasicRenderer.java:443)
at com.sun.faces.renderkit.html_basic.GridRenderer.encodeChildren(GridRenderer.java:233)
at javax.faces.component.UIComponentBase.encodeChildren(UIComponentBase.java:701)
at javax.faces.webapp.UIComponentTag.encodeChildren(UIComponentTag.java:607)
at javax.faces.webapp.UIComponentTag.doEndTag(UIComponentTag.java:544)
at com.sun.faces.taglib.html_basic.PanelGridTag.doEndTag(PanelGridTag.java:460)
at org.apache.jsp.login_jsp._jspx_meth_h_panelGrid_0(login_jsp.java:200)
at org.apache.jsp.login_jsp._jspx_meth_h_form_0(login_jsp.java:150)
at org.apache.jsp.login_jsp._jspx_meth_f_view_0(login_jsp.java:120)
at org.apache.jsp.login_jsp._jspService(login_jsp.java:85)
at org.apache.jasper.runtime.HttpJspBase.service(HttpJspBase.java:94)
at javax.servlet.http.HttpServlet.service(HttpServlet.java:802)
at org.apache.jasper.servlet.JspServletWrapper.service(JspServletWrapper.java:324)
at org.apache.jasper.servlet.JspServlet.serviceJspFile(JspServlet.java:292)
at org.apache.jasper.servlet.JspServlet.service(JspServlet.java:236)
at javax.servlet.http.HttpServlet.service(HttpServlet.java:802)
at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter
     (ApplicationFilterChain.java:237)
at org.apache.catalina.core.ApplicationFilterChain.doFilter
     (ApplicationFilterChain.java:157)
at org.apache.catalina.core.ApplicationDispatcher.invoke(ApplicationDispatcher.java:704)
at org.apache.catalina.core.ApplicationDispatcher.processRequest
     (ApplicationDispatcher.java:474)
at org.apache.catalina.core.ApplicationDispatcher.doForward
     (ApplicationDispatcher.java:409)
at org.apache.catalina.core.ApplicationDispatcher.forward
     (ApplicationDispatcher.java:312)
at com.sun.faces.context.ExternalContextImpl.dispatch(ExternalContextImpl.java:322)
at com.sun.faces.application.ViewHandlerImpl.renderView(ViewHandlerImpl.java:130)
at com.sun.faces.lifecycle.RenderResponsePhase.execute(RenderResponsePhase.java:87)
at com.sun.faces.lifecycle.LifecycleImpl.phase(LifecycleImpl.java:200)
at com.sun.faces.lifecycle.LifecycleImpl.render(LifecycleImpl.java:117)
at javax.faces.webapp.FacesServlet.service(FacesServlet.java:198)
at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter
     (ApplicationFilterChain.java:237)
at org.apache.catalina.core.ApplicationFilterChain.doFilter
     (ApplicationFilterChain.java:157)
at org.apache.catalina.core.StandardWrapperValve.invoke(StandardWrapperValve.java:214)
at org.apache.catalina.core.StandardValveContext.invokeNext
     (StandardValveContext.java:104)
at org.apache.catalina.core.StandardPipeline.invoke(StandardPipeline.java:520)
at org.apache.catalina.core.StandardContextValve.invokeInternal
     (StandardContextValve.java:198)
at org.apache.catalina.core.StandardContextValve.invoke(StandardContextValve.java:152)
at org.apache.catalina.core.StandardValveContext.invokeNext
     (StandardValveContext.java:104)
at org.apache.catalina.core.StandardPipeline.invoke(StandardPipeline.java:520)
at org.apache.catalina.core.StandardHostValve.invoke(StandardHostValve.java:137)
at org.apache.catalina.core.StandardValveContext.invokeNext
     (StandardValveContext.java:104)
at org.apache.catalina.valves.ErrorReportValve.invoke(ErrorReportValve.java:118)
at org.apache.catalina.core.StandardValveContext.invokeNext
     (StandardValveContext.java:102)
at org.apache.catalina.core.StandardPipeline.invoke(StandardPipeline.java:520)
at org.apache.catalina.core.StandardEngineValve.invoke(StandardEngineValve.java:109)
at org.apache.catalina.core.StandardValveContext.invokeNext
     (StandardValveContext.java:104)
at org.apache.catalina.core.StandardPipeline.invoke(StandardPipeline.java:520)
at org.apache.catalina.core.ContainerBase.invoke(ContainerBase.java:929)
at org.apache.coyote.tomcat5.CoyoteAdapter.service(CoyoteAdapter.java:160)
at org.apache.coyote.http11.Http11Processor.process(Http11Processor.java:799)
at org.apache.coyote.http11.Http11Protocol$Http11ConnectionHandler.processConnection
```

```
     (Http11Protocol.java:705)
  at org.apache.tomcat.util.net.TcpWorkerThread.runIt(PoolTcpEndpoint.java:577)
  at org.apache.tomcat.util.threads.ThreadPool$ControlRunnable.run(ThreadPool.java:683)
  at java.lang.Thread.run(Thread.java:534)
```

In previous articles in this series, starting with "Introducing Hamlets," I proposed an easily-used and easily-understood framework called *Hamlets* for the development of Web-based applications. The framework is the result of a radical software simplification effort. (You should read these older articles if you haven't already; you can find links in Resources.)

A Hamlet is a Java servlet extension that uses the Simple API for XML (SAX) to read template files. While a template file is being read, the Hamlet uses a small set of callback functions (implemented by a `HamletHandler`) to add dynamic content to those places in the template that are marked with special tags and IDs. Figure 1 illustrates the process.

Figure 1. A Hamlet uses SAX for reading content from a template file and calls a HamletHandler to add dynamic content

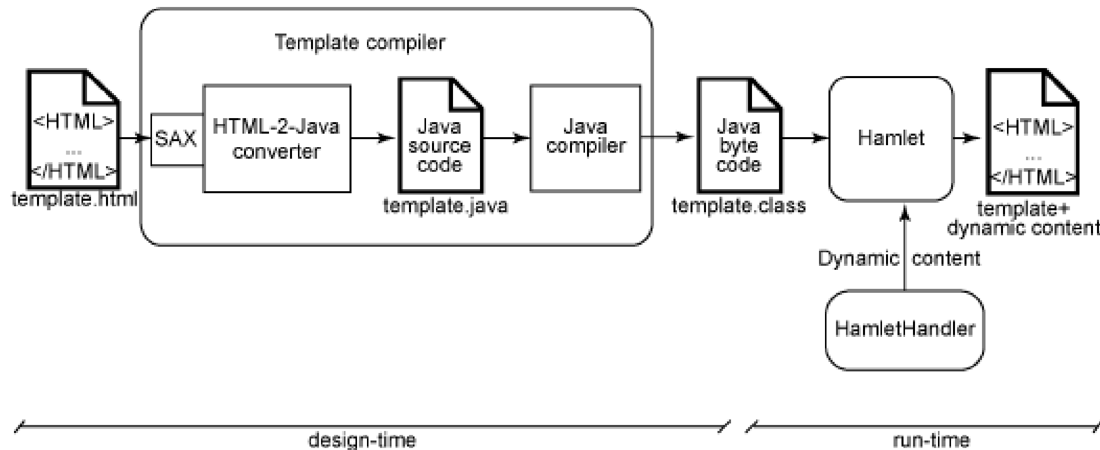


The project name *Hamlets* used in this publication refers to an internal project for the development of a servlet-based framework for separation of content and presentation in Web-based applications. You can download hamlet-compiler.zip, an archive containing the complete sample code from this article, from the Download section below.

Compiling Hamlet templates

This article describes a small addition to the Hamlet framework: a template compiler for accelerating Hamlets. No more than 600 lines of Java source code are necessary for its implementation. The template compiler uses SAX to read template files and converts the content into Java source code. It then calls the standard JDK Java compiler to generate Java bytecode. At runtime, the Hamlet executes this code and calls a `HamletHandler` to add dynamic content. Figure 2 illustrates the process.

Figure 2. A template compiler converts the content of a template file into Java bytecode. The Hamlet executes this code and calls a HamletHandler to add dynamic content.



Compiled Hamlets (that is, Hamlets that use compiled templates) allow the implementation of Web-based applications with a large number of concurrent users. But they are equally suitable for embedded devices due to their modest resource requirements.

How it works

The template compiler operates in two stages. First, the HTML content of a template file is read and converted into Java source code. For example, consider Listing 2: it includes the HTML content in BasicTestTemplate.html.

Listing 2. Basic HTML code

```
<HTML>
  <HEAD>
    <TITLE>Hamlet Test Suite</TITLE>
  </HEAD>
  <BODY>

    <H1>Hamlet Test Suite</H1>
    <DIV>
      The following test cases test the functionality of the <I>hamlet.jar</I> library.
      It is in working condition if the output in each section is exactly repeated.
    </DIV>

    <H2>Replace Test</H2>
    <DIV>Hello Mr. <REPLACE ID="Name">X</REPLACE>.</DIV>
    <DIV>Hello Mr. Pawlitzek.</DIV>

    <H2>Repeat Test</H2>
    <REPEAT ID="Rows">
      <DIV>
      <REPEAT ID="Cols">
        <REPLACE ID="Value">0</REPLACE>
      </REPEAT>
      </DIV>
    </REPEAT>
    <DIV> </DIV>
    <DIV>11 12 13</DIV>
    <DIV>21 22 23</DIV>
    <DIV>31 32 33</DIV>
```

```
    <H2>Attribute Test</H2>
    <DIV><A>www.pawlitzek.com</A></DIV>
    <DIV><A ID="Ref" HREF="www.pawlitzek.com">www.pawlitzek.com</A></DIV>

    <H2>Include Test</H2>
    <DIV>© Copyright IBM Corp. 2006</DIV>
    <DIV><INCLUDE SRC="BasicTestInclude.html" /></DIV>
    <DIV><INCLUDE ID="Copyright" SRC="BasicTestInclude.html" /></DIV>

  </BODY>
</HTML>
```

This code is converted into the Java source code (BasicTestTemplate.java) in Listing 3.

Listing 3. HTML from Listing 2, converted into Java code

```
import java.io.*;
import com.ibm.hamlet.*;
import org.xml.sax.helpers.*;

public class BasicTestTemplate implements Template {

  public void serveDoc (PrintWriter writer, ContentHandler handler) throws Exception {
    writer.print (
      "<HTML>\n" +
      "  <HEAD>\n" +
      "   <TITLE>Hamlet Test Suite</TITLE>\n" +
      "  </HEAD>\n" +
      "  <BODY>\n" +
      "\n" +
      "   <H1>Hamlet Test Suite</H1>\n" +
      "   <DIV>\n" +
      "    The following test cases test the functionality" +
      " of the <I>hamlet.jar</I> library.\n" +
      "    It is in working condition if the output in each section" +
      " is exactly repeated.\n" +
      "   </DIV>\n" +
      "\n" +
      "   <H2>Replace Test</H2>\n" +
      "   <DIV>Hello Mr. "
    );
    AttributesImpl atts0 = new AttributesImpl ();
    atts0.addAttribute ("", "ID", "ID", "CDATA", "Name");
    writer.print (handler.getElementReplacement ("Name", "REPLACE", atts0));
    writer.print (
      ".</DIV>\n" +
      "   <DIV>Hello Mr. Pawlitzek.</DIV>\n" +
      "\n" +
      "   <H2>Repeat Test</H2>\n" +
      "    "
    );
    AttributesImpl atts1 = new AttributesImpl ();
    atts1.addAttribute ("", "ID", "ID", "CDATA", "Rows");
    int count0 = handler.getElementRepeatCount ("Rows", "REPEAT", atts1);
    for (int loop0 = 0; loop0 < count0; loop0++) {
      writer.print (
        "\n" +
        "      <DIV>\n" +
        "      "
      );
      AttributesImpl atts2 = new AttributesImpl ();
      atts2.addAttribute ("", "ID", "ID", "CDATA", "Cols");
      int count1 = handler.getElementRepeatCount ("Cols", "REPEAT", atts2);
      for (int loop1 = 0; loop1 < count1; loop1++) {
```

```
        writer.print (
          "\n" +
          "        "
        );
        AttributesImpl atts3 = new AttributesImpl ();
        atts3.addAttribute ("", "ID", "ID", "CDATA", "Value");
        writer.print (handler.getElementReplacement ("Value", "REPLACE", atts3));
        writer.print (
          "\n" +
          "      "
        );
        if (handler.getElementRepeatCount ("Cols", "REPEAT", atts2) == 0)
          break;
      } // for
      writer.print (
        "\n" +
        "      </DIV>\n" +
        "    "
      );
      if (handler.getElementRepeatCount ("Rows", "REPEAT", atts1) == 0)
        break;
    } // for
    writer.print (
      "\n" +
      "   <DIV> </DIV>\n" +
      "   <DIV>11 12 13</DIV>\n" +
      "   <DIV>21 22 23</DIV>\n" +
      "   <DIV>31 32 33</DIV>\n" +
      "\n" +
      "   <H2>Attribute Test</H2>\n" +
      "   <DIV><A>www.pawlitzek.com</A></DIV>\n" +
      "   <DIV>"
    );
    AttributesImpl atts4 = new AttributesImpl ();
    atts4.addAttribute ("", "ID", "ID", "CDATA", "Ref");
    atts4.addAttribute ("", "HREF", "HREF", "CDATA", "www.pawlitzek.com");
    RuntimeUtilities.printTag
          (writer, "A", handler.getElementAttributes ("Ref", "A", atts4));
    writer.print (
      "www.pawlitzek.com</A></DIV>\n" +
      "\n" +
      "   <H2>Include Test</H2>\n" +
      "   <DIV>(c) Copyright IBM Corp. 2006</DIV>\n" +
      "   <DIV>"
    );
    AttributesImpl atts5 = new AttributesImpl ();
    atts5.addAttribute ("", "SRC", "SRC", "CDATA", "BasicTestInclude.html");
    RuntimeUtilities.printInclude
          (writer, handler.getElementIncludeSource (null, "INCLUDE", atts5));
    writer.print (
      "</DIV>\n" +
      "   <DIV>"
    );
    AttributesImpl atts6 = new AttributesImpl ();
    atts6.addAttribute ("", "ID", "ID", "CDATA", "Copyright");
    atts6.addAttribute ("", "SRC", "SRC", "CDATA", "BasicTestInclude.html");
    RuntimeUtilities.printInclude (writer,
      handler.getElementIncludeSource ("Copyright", "INCLUDE",
        handler.getElementAttributes ("Copyright", "INCLUDE", atts6)));
    writer.print (
      "</DIV>\n" +
      "\n" +
      "  </BODY>\n" +
      "</HTML>"
    );
  } // serveDoc
```

```
} // BasicTestTemplate
```

The conversion is straightforward. The following transformations are performed:

- Sections of static content are collected, and for each section a `writer.print()` statement is generated. For example, the code in Listing 4 is transformed into the code in Listing 5.

  Listing 4. Static content sections

```
<HTML>
  <HEAD>
    <TITLE>Hamlet Test Suite</TITLE>
  </HEAD>
  <BODY>

    <H1>Hamlet Test Suite</H1>
    <DIV>
      The following test cases test the functionality of the <I>hamlet.jar</I> library.
      It is in working condition if the output in each section is exactly repeated.
    </DIV>

    <H2>Replace Test</H2>
    <DIV>Hello Mr.
```

  Listing 5. HTML from Listing 4, converted into Java code

```
  writer.print (
    "<HTML>\n" +
    "  <HEAD>\n" +
    "   <TITLE>Hamlet Test Suite</TITLE>\n" +
    "  </HEAD>\n" +
    "  <BODY>\n" +
    "\n" +
    "   <H1>Hamlet Test Suite</H1>\n" +
    "   <DIV>\n" +
    "    The following test cases test the functionality" +
    " of the <I>hamlet.jar</I> library.\n" +
    "    It is in working condition if the output in each section" +
    " is exactly repeated.\n" +
    "   </DIV>\n" +
    "\n" +
    "   <H2>Replace Test</H2>\n" +
    "   <DIV>Hello Mr. "
  );
```

- For each `<REPLACE>...</REPLACE>` section, a `getElementReplacement()` and a `writer.print()` call are generated. At runtime, the Hamlet invokes `getElementReplacement()` with the attributes of the `<REPLACE>` tag to receive dynamic content that is then included in the output by `writer.print()`. For example, the code in Listing 6 is transformed into the code in Listing 7.

  Listing 6. <REPLACE> section

```
<REPLACE ID="Name">X</REPLACE>
```

Listing 7. HTML from Listing 6, converted into Java code

```
AttributesImpl atts0 = new AttributesImpl ();
atts0.addAttribute ("", "ID", "ID", "CDATA", "Name");
writer.print (
  handler.getElementReplacement ("Name", "REPLACE", atts0));
```

- For each `<REPEAT>...</REPEAT>` section, a `for` loop statement and two `getElementRepeatCount()` calls are generated. At runtime, the Hamlet invokes `getElementRepeatCount()` with the attributes of the `<REPEAT` tag: once before the loop is executed to obtain the number of repeats, and repeatedly within the loop to allow termination by returning 0. For example, the code in Listing 8 is transformed into the code in Listing 9.

Listing 8. <REPEAT> section

```
<REPEAT ID="Cols">
  ...
</REPEAT>
```

Listing 9. HTML from Listing 8, converted into Java code

```
AttributesImpl atts2 = new AttributesImpl ();
atts2.addAttribute ("", "ID", "ID", "CDATA", "Cols");
int count1 = handler.getElementRepeatCount ("Cols", "REPEAT", atts2);
for (int loop1 = 0; loop1 < count1; loop1++) {
  ...
  if (handler.getElementRepeatCount ("Cols", "REPEAT", atts2) == 0)
    break;
} // for
```

- For each `<INCLUDE/>` section, a `getElementIncludeSource()` and a `RuntimeUtilities.printInclude()` call are generated. At runtime, the Hamlet invokes `getElementIncludeSource()` with the attributes of the `<INCLUDE>` tag to obtain an `InputStream`. Next, `RuntimeUtilities.printInclude()` uses this `InputStream` to include the content of the resource that is specified with the `SRC` attribute of the `<INCLUDE>` tag. For example, the code in Listing 10 is transformed into the code in Listing 11.

Listing 10. <INCLUDE/> section

```
<INCLUDE SRC="BasicTestInclude.html" />
```

Listing 11. HTML from Listing 10, converted into Java code

```
AttributesImpl atts5 = new AttributesImpl ();
atts5.addAttribute ("", "SRC", "SRC", "CDATA", "BasicTestInclude.html");
RuntimeUtilities.printInclude (writer,
  handler.getElementIncludeSource (null, "INCLUDE", atts5));
```

- For each `<INCLUDE ID="..."/>` section, a `getElementAttributes()`, a `getElementIncludeSource()`, and a `RuntimeUtilities.printInclude()` call are generated. At runtime, the Hamlet first invokes `getElementAttributes()` with the attributes of the `<INCLUDE>` tag to allow attribute modifications. Next, `getElementIncludeSource()` is called with the possibly modified attributes to

obtain an `InputStream`. Finally, `RuntimeUtilities.printInclude()` uses this `InputStream` to include the content of a resource. For example, the code in Listing 12 is transformed into the code in Listing 14.

Listing 12. <INCLUDE ID="..."/> section

```
<INCLUDE ID="Copyright" SRC="BasicTestInclude.html" />
```

Listing 13. HTML from Listing 12, converted into Java code

```
AttributesImpl atts6 = new AttributesImpl ();
atts6.addAttribute ("", "ID", "ID", "CDATA", "Copyright");
atts6.addAttribute ("", "SRC", "SRC", "CDATA", "BasicTestInclude.html");
RuntimeUtilities.printInclude (writer,
  handler.getElementIncludeSource ("Copyright", "INCLUDE",
    handler.getElementAttributes ("Copyright", "INCLUDE", atts6)));
```

- For each tag with an `ID` attribute, a `getElementAttributes()` and a `RuntimeUtilities.printTag()` call are generated. At runtime, the Hamlet invokes `getElementAttributes()` with the tag's attributes to allow attribute modifications. Subsequently, `RuntimeUtilities.printTag()` is called to include the tag with its possibly modified attributes in the output. For example, the code in Listing 14 is transformed into the code in Listing 15.

Listing 14. Tag with an ID attribute

```
<A ID="Ref" HREF="www.pawlitzek.com">
```

Listing 15. HTML from Listing 14, converted into Java code

```
AttributesImpl atts4 = new AttributesImpl ();
atts4.addAttribute ("", "ID", "ID", "CDATA", "Ref");
atts4.addAttribute ("", "HREF", "HREF", "CDATA", "www.pawlitzek.com");
RuntimeUtilities.printTag (writer, "A",
  handler.getElementAttributes ("Ref", "A", atts4));
```

This is stage one of the template compiler: the conversion of the HTML template into Java source code. In stage two, the standard JDK Java compiler is called to compile the generated source from stage one. The resulting Java bytecode is executed at runtime by the Hamlet and calls the four callback functions (`getElementReplacement()`, `getElementRepeatCount()`, `getElementAttributes()`, and `getElementIncludeSource()`) implemented by the `HamletHandler` to add dynamic content to the response.

Implementing the template compiler

Starting the template compiler from the command line invokes its static `main()` function with the command-line arguments, as in Listing 16.

Listing 16. The static main() function creates an instance of the TemplateCompiler class and calls its perform() method

```
public class TemplateCompiler {

  ...
```

```
  public static void main (String args[]) {
    try {
      if (args.length == 0) {
        System.out.println (
         "Usage: java -cp <classpath> com.ibm.hamlet.compiler.TemplateCompiler " +
         "[-debug] [-verbose] [-dstDir <directory>] <Template.html>");
        Runtime.getRuntime().exit (1);
      } // if
      TemplateCompiler tc = new TemplateCompiler ();
      for (int n = 0; n < args.length; n++) {
        if ("-debug".equals (args[n]))
          tc.setDebug (true);
        else if ("-verbose".equals (args[n]))
          tc.setVerbose (true);
        else if ("-dstDir".equals (args[n]))
          tc.setDstDir (args[++n]);
        else
          tc.setFileName (args[n]);
      } // for
      tc.perform ();
    } catch (Exception e) {
      e.printStackTrace ();
      Runtime.getRuntime().exit (1);
    } // try
  } // main

} // TemplateCompiler
```

First, the compiler checks the number of arguments. If no arguments are present, a short message on how to use the template compiler is printed and the application terminates. Otherwise, an instance of the `TemplateCompiler` class is created (`tc`) and the command-line arguments are parsed. Next, the template compiler's `perform()` method is called to do the actual work. It implements the two stages described above -- the conversion of the HTML template into Java source code, and the compilation of that source code -- as in Listing 17.

Listing 17. perform() implements the two stages of the HTML-to-Java bytecode conversion

```
public class TemplateCompiler {

  ...

  public void perform () throws Exception {

    // stage one: create template.java
    File fin = new File (fileName);
    String templateName = fin.getName ();
    String className = RuntimeUtilities.getClassName (templateName);
    String sourceName = dstDir + File.separator + className + ".java";
    File fout = new File (sourceName);
    InputStream in = new FileInputStream (fin);
    OutputStream out = new FileOutputStream (fout);
    SourceGenerator generator = getSourceGenerator ();
    if (verbose) System.out.println ("Creating " + sourceName + " ...");
    generator.perform (in, out, className);
    out.close ();
    in.close ();

    // stage two: compile template.java
    String[] args = { sourceName };
    Main compiler = new Main ();
```

```
    if (verbose) System.out.println ("Compiling " + sourceName + " ...");
    compiler.compile (args);
    // delete template.java
    if (!debug) {
      if (verbose) System.out.println ("Deleting " + sourceName + " ...");
      fout.delete ();
    } // if
    if (verbose) System.out.println ("Finished.");

  } // perform

  ...

} // TemplateCompiler
```

In stage one, a `FileInputStream` and a `FileOutputStream` are constructed before `getSourceGenerator()` obtains an object (`generator`) that implements the `SourceGenerator` interface. Listing 18 illustrates this interface.

Listing 18. The SourceGenerator interface

```
public interface SourceGenerator {

  public void perform (InputStream in, OutputStream out, String name)
    throws Exception;

} // SourceGenerator
```

By calling `generator.perform (in, out, className)`, the HTML template is read from the input stream, converted into Java source code, and written to the output stream.

In stage two, an instance of the standard JDK Java compiler generates Java bytecode from the Java source code produced in stage one when `compiler.compile (args)` is executed. Afterwards, the template compiler deletes the no longer used Java source if the debug mode is not activated.

Source code generation

The `DefaultGenerator` class converts the HTML template into Java code, as Listing 19 illustrates. This class implements the `SourceGenerator` interface.

Listing 19. The DefaultGenerator's perform() method initiates the conversion of HTML into Java source code

```
public class DefaultGenerator extends DefaultHandler implements SourceGenerator {

  ...

  public void perform (InputStream in, OutputStream out, String source)
    throws Exception {

    XMLReader reader = null;
    try {
      this.source = source;
      this.out = new PrintStream (out);
      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

  ...

} // DefaultGenerator
```

The `DefaultGenerator`'s `perform()` method obtains a SAX reader from a pool of readers, sets the SAX error and content handlers, and starts to parse the input source (the template). The SAX reader returns to the pool of readers when the parsing is finished.

During the parsing of a template, the generator's SAX content handler methods (`startDocument()`, `endDocument()`, `startElement()`, `endElement()`, and `characters()`) are called. Listing 20 illustrates the implementation.

Listing 20. The DefaultGenerator class implements the SAX content handler methods that convert HTML into Java source code

```
public class DefaultGenerator extends DefaultHandler implements SourceGenerator {

  ...

  public void startDocument () throws SAXException {
    stack = new Stack ();
    buf = new StringBuffer ();
    generateHeader (out, source);
  } // startDocument


  public void endDocument () throws SAXException {
    generateText (out, buf.toString ());
    generateFooter (out, source);
  } // endDocument


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

    // category.debug ("local name: " + localName + ", qname: " + qName);
    if (REPEAT_TAG.equals (localName)) {
      generateText (out, buf.toString ());
      generateElementRepeatHeader (out, atts);
      buf = new StringBuffer ();
    } else if (REPLACE_TAG.equals (localName)) {
      generateText (out, buf.toString ());
      String id = atts.getValue (ID_ATTRIBUTE);
      if (id != null)
        generateElementReplacement (out, atts);
      buf = new StringBuffer ();
      ignore = true;
    } else if (INCLUDE_TAG.equals (localName)) {
      generateText (out, buf.toString ());
      generateElementInclude (out, localName, atts);
      buf = new StringBuffer ();
      ignore = true;
```

```
    } else {
      String id = atts.getValue (ID_ATTRIBUTE);
      if (id != null) {
        generateText (out, buf.toString ());
        generateElementAttributes (out, localName, atts);
        buf = new StringBuffer ();
      } else {
        buf.append ("<");
        buf.append (localName);
        for (int i = 0; i < atts.getLength (); i++) {
          buf.append (" ");
          buf.append (atts.getLocalName (i));
          buf.append ("=\"");
          buf.append (atts.getValue (i));
          buf.append ("\"");
        } // for
        buf.append (">");
      } // if
    } // if

  } // startElement


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

    if (REPEAT_TAG.equals (localName)) {
      generateText (out, buf.toString ());
      generateElementRepeatFooter (out);
      buf = new StringBuffer ();
    } else if (REPLACE_TAG.equals (localName) || INCLUDE_TAG.equals (localName)) {
      ignore = false;
    } else {
      buf.append ("</");
      buf.append (localName);
      buf.append (">");
    } // if

  } // endElement


  public void characters (char[] ch, int start, int length) throws SAXException {
    chars = new String (ch, start, length);
    if (!ignore)
      buf.append (chars);
  } // characters

  ...

} // DefaultGenerator
```

The `startDocument()` method is called before the template is actually parsed. It creates a `StringBuffer` to collect static template content, and a stack for handling nested `<REPEAT>` tags. Furthermore, `generateHeader(out, source)` is called to generate the source code header, which contains import statements and the `serveDoc()` method definition.

The SAX parser invokes `startElement()` and `endElement()` whenever it recognizes starting and ending tags. These methods convert the template content according to the description outlined in the section above entitled "How it works."

For example, when the SAX reader comes across a `<REPEAT>` tag, `startElement()` flushes the current `StringBuffer`. In other words, `generateText(out, buf.toString())` generates a `writer.print()` statement in the Java source code. The argument of the `writer.print()` statement is a string containing all the collected static HTML content that was read in until the `<REPEAT>` tag was encountered. Next, `generateElementRepeatHeader(out, atts)` produces the source for the beginning of a `for` loop. And finally, a new `StringBuffer` instance is created and parsing continues. Listing 21 illustrates all this.

Listing 21. <REPEAT> tag processing in startElement() generates Java source code for the beginning of a for loop

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

  if (REPEAT_TAG.equals (localName)) {
    generateText (out, buf.toString ());
    generateElementRepeatHeader (out, atts);
    buf = new StringBuffer ();
  } ...

} // startElement
```

When the SAX reader comes across the corresponding `</REPEAT>` tag, `endElement()` flushes the current `StringBuffer` and calls `generateElementRepeatFooter(out)` to complete the Java source code for the previously unfinished `for` loop. Again, a new `StringBuffer` is created before parsing goes on. Listing 22 illustrates this.

Listing 22. </REPEAT> tag processing in endElement() generates Java source code to complete the unfinished for loop

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

  if (REPEAT_TAG.equals (localName)) {
    generateText (out, buf.toString ());
    generateElementRepeatFooter (out);
    buf = new StringBuffer ();
  } ...

} // endElement
```

`<REPLACE>` and `<INCLUDE>` tags cause code generation in much the same way. This is also true for all other tags that include an `ID` attribute.

The `endDocument()` method is called when the parsing has ended. It flushes the `StringBuffer` and generates the source code footer with `generateFooter(out, source)`.

And this is how HTML templates are compiled into Java classes at development time.

Executing the compiled templates

At runtime, when an application server or a servlet container receives a user request, the Hamlet's `doGet()` method is invoked. Within `doGet()`, a Hamlet creates a *handler,* an instance of a class that extends `HamletHandler` and that provides default implementations for the four callback methods: `getElementReplacement()`, `getElementRepeatCount()`, `getElementAttributes()`, and `getElementIncludeSource()`. A class that extends `HamletHandler` (`BasicTestHandler`, for instance) overwrites the four callback methods to provide specific dynamic content for a particular Hamlet, as in Listing 23.

Listing 23. An instance of the BasicTestHandler class provides dynamic content for BasicTestTemplate.html

```
public class BasicTest extends Hamlet {

  ...

  private static class BasicTestHandler extends HamletHandler {

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

    public int getElementRepeatCount (String id, String name, Attributes atts)
      throws Exception {
      ...
    } // getElementRepeatCout

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

    public InputStream getElementIncludeSource (String id, String name, Attributes atts)
      throws Exception {
      ...
    } // getElementIncludeSource

  } // BasicTestHandler


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

    try {
      HamletHandler handler = new BasicTestHandler (this);
      serveDoc (req, res, "BasicTestTemplate.html", handler);
    } catch (Exception e) {
      category.error ("", e);
      throw new ServletException (e);
    } // try

  } // doGet

} // BasicTest
```

Subsequently, the Hamlet's `serveDoc()` method is called and the template's name and the handler are passed as arguments (together with the servlet request (`req`) and

servlet response (`res`)). `serveDoc()` itself calls the private `serveDoc()` method as in Listing 24.

Listing 24. The Hamlet's private serveDoc() method invokes findTemplateClass() to find the Java bytecode of the compiled template

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

  ...

  private void serveDoc (PrintWriter out, String template, ContentHandler handler)
    throws Exception {

    findTemplateClass (template);
    if (templateClass != null) {
      ...
    } else {
      ...
    } // if

  } // serveDoc

} // Hamlet
```

This method invokes `findTemplateClass(template)` to find the Java bytecode of the compiled template. Listing 25 shows the implementation of `findTemplateClass()`.

Listing 25. The findTemplateClass() method loads the Java bytecode of the template

```
  private void findTemplateClass (String template) throws Exception {
    if (!template.equals (oldTemplate)) {
      String className = RuntimeUtilities.getClassName (template);
      try {
        category.debug ("Loading class '" + className + "' ...");
        Class c = Class.forName (className);
        templateClass = (Template) c.newInstance ();
        category.debug ("Class '" + className + "' loaded");
      } catch (ClassNotFoundException e) {
        category.debug ("Cannot load class '" + className + "'");
      } // try
      oldTemplate = template;
    } // if
  } // findTemplateClass
```

`findTemplateClass()` checks whether the Java bytecode of a template needs to be loaded. If so, `RuntimeUtilities.getClassName(template)` returns the name of a class (`className`) that contains the compiled Java source code of the template. With `Class.forName(className)`, the class is loaded before an instance (`templateClass`) is created. Listing 26 illustrates the instance is of type `Template`.

Listing 26. The template interface defines the interface of a template class

```
  public interface Template {

    public void serveDoc (PrintWriter writer, ContentHandler handler) throws Exception;

  } // Template
```

The template compiler generates a separate Java class for each HTML template. Each of these classes implements the `Template` interface and therefore provides a `serveDoc()` method, as in Listing 27.

Once the class containing the compiled Java code of the template is found, it is run by calling `templateClass.serveDoc(out, handler)`. If it is not found, the Hamlet falls back into the *non-compiled mode* -- in other words, it obtains a template engine that parses the template using SAX (as it always did until Hamlet compilation was invented).

Listing 27. The Hamlet's serveDoc() method executes the Java bytecode of the template if it exists; otherwise, a template engine parses the template on the fly

```
private void serveDoc (PrintWriter out, String template, ContentHandler handler)
  throws Exception {

  findTemplateClass (template);
  if (templateClass != null) {
    templateClass.serveDoc (out, handler);
  } else {
    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.");
  } // if

} // serveDoc
```

To be or not to be compiled? The request is fulfilled in both cases. If you use a compiled template, you see a noticeable boost in performance results. The scale of the gain depends primarily on the amount of static content in the template. For very small templates with about 50 tags (like BasicTestTemplate.html), a compiled Hamlet is 1.2 times faster. Larger templates with about 1,000 tags execute 1.75 times more quickly when compiled.

Calling the template compiler from Ant

Template compilation takes place at development time. Therefore, it is convenient to have an Ant task like the one in Listing 28 in the build script to invoke the template compiler.

Listing 28. Call the template compiler from an Ant script

```
<project name="build" default="jar" basedir=".">

  ...

  <target name="template" depends="compile">
    <taskdef name="hamletc" classname="com.ibm.hamlet.ant.CompileTask"/>
    <hamletc destdir="${dst}">
      <fileset dir="${src}">
        <include name="*Template.html" />
      </fileset>
    </hamletc>
  </target>

</project>
```

Listing 29 shows how to implement a basic Ant task.

Listing 29. The CompileTask class implements an Ant task to invoke the template compiler

```
public class CompileTask extends Task {

  ...

  private  String     dstDir;
  private  ArrayList  fileSets = new ArrayList ();


  public void setDestdir (String dir) {
    dstDir = dir;
    System.out.println ("Destination directory: " + dstDir);
  } // setDestdir


  public void addFileset (FileSet set) {
    fileSets.add (set);
  } // addFileset


  public void execute () {
    // setup compiler
    TemplateCompiler compiler = new TemplateCompiler ();
    compiler.setDebug (debug);
    compiler.setVerbose (verbose);
    compiler.setDstDir (dstDir);
    // invoke compiler for all files
    Project project = getProject ();
    Iterator iter = fileSets.iterator ();
    while (iter.hasNext ()) {
      Object elem = iter.next ();
      if (elem instanceof FileSet) {
        FileSet set = (FileSet) elem;
        DirectoryScanner scanner = set.getDirectoryScanner (project);
        String srcDir = scanner.getBasedir().getPath ();
        System.out.println ("Source directory: " + srcDir);
        String fileNames[] = scanner.getIncludedFiles ();
        System.out.println ("Compiling " + fileNames.length +
          " template file(s) to " + dstDir);
        for (int i = 0; i < fileNames.length; i++) {
          String fileName = srcDir + File.separator + fileNames[i];
          // System.out.println (" scanning " + fileName);
          compiler.setFileName (fileName);
          try {
```

```
            compiler.perform ();
          } catch (Exception e) {
            throw new BuildException (e);
          } // try
        } // for
      } // if
    } // while
  } // execute

} // CompileTask
```

Executing the Ant task shown in Listing 28 creates an instance of the `com.ibm.hamlet.ant.CompileTask` class and invokes its `execute()` method, after setting the destination directory and specifying the file set to be compiled with `setDestdir()` and `addFileset()`, respectively. Within `execute()`, a template compiler is instantiated in order to compile all template files of the file set to the destination directory.

In conclusion

In this article, I introduced a small addition to the Hamlet framework: a template compiler to convert template files into Java bytecode. The resulting code is executed by the Hamlet to fulfill user requests at runtime. Hamlets that use such compiled templates (so-called *compiled Hamlets*) offer a noticeable boost in performance and thus enable the development of Web-based applications with a large number of concurrent users.

The complete Java source code for the template compiler (less than 600 lines) is available from Download.

Downloads

Description	Name	Size	Download method
Complete Java source code for template compiler	wa-hamlets4/hamlet-compiler.zip	6KB	HTTP

Information about download methods

Resources

Learn

- "Introducing Hamlets," René Pawlitzek (developerWorks, March 2005): Get the fundamentals of Hamlet programming and learn how to separate content and presentation with just a smidgeon of code.
- "Programming Hamlets," René Pawlitzek (developerWorks, May 2005): In this tutorial, learn various aspects of Hamlet v1.0 programming and review several practical Hamlet examples.
- "Implementing Hamlets," René Pawlitzek (developerWorks, February 2006): Read about the Hamlet v1.1 implementation and `HamletHandler`s, an additional way to provide dynamic content.
- The Java Servlet API White Paper: Get an overview of Java servlets.
- developerWorks Web Architecture zone: Expand your site development skills with articles and tutorials that specialize in Web technologies.
- developerWorks technical events and webcasts: Stay current with jam-packed technical sessions that shorten your learning curve, and improve the quality and results of your most difficult software projects.

Get products and technologies

- SAX, the Simple API for XML: Check out this widely used API.
- Ant: Find out more about this Java technology-based build tool.
- Hamlets on SourceForge: Now that this project is open source (as of June 2007), find out how you can contribute.
- IBM trial software: Build your next development project with software available for download directly from developerWorks.

Discuss

- Participate in the discussion forum for this content.
- developerWorks blogs: Get involved in the developerWorks community.
- developerWorks discussion forums: Join the discussion threads that interest you.

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.

© Copyright IBM Corporation 2006
(www.ibm.com/legal/copytrade.shtml)
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)