Introducing Hamlets

A small-footprint, servlet-based, content creation framework

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

22 March 2005

Servlets are a key component of server-side Java™ development, but despite a number of attractive traits, servlets do not support or enforce the separation of content and presentation. To master that functionality, René Pawlitzek proposes Hamlets -- servlet extensions that provide this functionality within a lightweight framework implemented with less than 500 lines of Java source code.

Server-side Java development has increased in popularity in the last few years and servlets are a key part of it. A servlet is a small, pluggable extension to a Web or application server that provides its capabilities in a Java class. The servlet is loaded at runtime to expand the server's functionality.

Servlets, an ideal choice for Web development, have a number of assets -- portability, efficiency, safety, extensibility, and flexibility. Few viable alternatives exist that can match the power and elegance of servlets (a widespread, competing technology is Microsoft's Active Server Pages, or ASP).

One problem, though, exists with servlets when it comes to Web development: They do not enforce or support the separation of content and presentation. Why is this separation important? (I offer reasons in the next section.)

This article introduces Hamlets, servlet extensions that provide this functionality within a lightweight framework implemented with less than 500 lines of Java source code.

Mix Java and HTML in the same page?

Despite their attractive properties, out-of-the-box servlets are missing an important feature: support for the separation of content from presentation. If servlets are exclusively used for the development of Web-based applications, Java and HTML code inevitably ends up intermixed in the same source file (see Listing 1).

Listing 1. "Hello World" servlet with embedded HTML code

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloWorld extends HttpServlet {

  public void doGet (HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {
    res.setContentType ("text/html");
    PrintWriter out = res.getWriter ();
    out.println ("<HTML>">
    out.println ("<HEAD><TITLE>Hello World</TITLE></HEAD>");
    out.println ("<BODY>");
    out.println ("Hello World");
    out.println ("</BODY>");
    out.println ("</HTML>");
  } // doGet

} // HelloWorld
```

The separation of Java and HTML code is highly desirable because:

- Developers and designers can work independently of each other on the Java and the HTML code.
- The ease of maintaining the application increases.
- The opportunity to introduce errors is smaller since a change in the HTML code does not require retesting the Java code.

Several servlet-based frameworks have evolved to support the separation of content from presentation (Table 1 offers a partial overview).

Table 1. A few frameworks that support the separation of content from presentation

Framework	Concept	Verdict
JavaServer Pages (JSP)	A JavaServer Page™ can contain both HTML and Java code. However, placing Java code within a JSP is considered bad practice because content and presentation are intermixed rather than separated. Use JavaServer Pages primarily for presentation logic. With the help of JavaBeans and custom tag libraries, you can eliminate Java code from JSPs.	JavaServer Pages enable the separation of content from presentation, but don't enforce or encourage it as other alternatives do. The compiler that translates a JSP into a background servlet introduces a first-person penalty and a security risk.
WebMacro	WebMacro is a template engine. It performs text replacement operations inside Web page templates. Servlets are used to push content into templates.	WebMacro offers good separation of content and presentation, but requires a (simplified) scripting syntax. It is suitable for Web applications with high functionality.
Tea	Developers create applications in Java code and install them in the framework. To create and maintain the final appearance of dynamic Web pages, producers write Tea templates that call functions provided by the installed applications.	The templates (written in the Tea language) enforce the separation of content from presentation, but the Tea language adds additional complexity. Tea hides the servlet infrastructure. Less programmer involvement is required.
XML Compiler (XMLC)	XMLC compiles HTML documents into a Java class that contains instructions to create an XML DOM tree representation of the document	XMLC achieves a high level of separation between content and presentation, but DOM trees require many resources.

	in memory. The developer writes code that manipulates the tree to add dynamic content before it is output.	
Element Construction Set (ECS)	With ECS, you generate markup code with Java objects.	ECS hides markup code in Java objects. In many cases, it is too programmer-centric.

The initial motivation to provide an alternative framework came from the fact that none of these frameworks seemed suitable for the development of a Web-based console to monitor intrusion-detection events for various reasons (license, complexity, performance, security, and so on). Note that the applicability of the newly created framework is not limited to the area of intrusion detection.

What does a new framework need?

Before creating a new framework, I compiled a list of requirements that is actually rather short. These three basic requirements are important:

- The framework not only supports, but enforces, the complete separation of content from presentation. Only designers work on the presentation using HTML and only developers provide the dynamic content using Java technology.
- The framework has a simple and elegant design that does not hide the familiar underlying servlet infrastructure.
- The framework is implemented in a lightweight manner with minimal overhead so that it is easy to use and understand.

Meet Hamlets

With these requirements in mind, I devised a small-footprint (on the order of 500 lines of Java source code) servlet-based, content creation framework called Hamlets. One can describe a Hamlet as follows:

A Hamlet is a servlet extension that reads XHTML template files containing presentation using SAX (the Simple API for XML) and dynamically adds content on the fly to those places in the template which are marked with special tags and IDs using a small set of callback functions.

Hamlets support the separation of responsibilities in combination with other Web technologies. Consider the following set-up for Web application development:

- Hamlets provide the code which creates the dynamic content in a Web application and can work with Java Beans that contain the business logic for further code structuring. Developers write Hamlets in Java code.
- XHTML template files contain mock-up pages that provide the presentation in a Web application. In other words, they contain the basic layout of a dynamic Web page. Web designers create the template files.
- Cascading Style Sheets (CSS) format Web pages and provide a consistent look and feel across projects and Web pages. Web designers also create the stylesheet files.

Program Hamlets

You need these programming elements to make Hamlets work.

<REPLACE> and getElementReplacement()

The <REPLACE> tag and the `getElementReplacement()` callback comprise the mechanism to create dynamic content within the Hamlet framework. After a Web designer finishes the XHTML code of a Web page, the <REPLACE> tag (together with an ID attribute) marks all those places within the XHTML file where the Hamlet framework will insert dynamic content during a request. The designer does not know the actual value during the design phase and uses placeholder values instead: `<REPLACE ID="time">12:34</REPLACE>`.

Once a Hamlet receives a user request, its `doGet()`/`doPost()` method is invoked. This function passes the name of the XHTML template file to the `serveDoc()` method of the Hamlet framework where the actual work is done.

```
public synchronized void doGet (HttpServletRequest req, HttpServletResponse res)
  throws ServletException {
  try {
    // serve document
    serveDoc (req, res, "ClockTemplate.html");
  } catch (Exception e) {
    throw new ServletException (e);
  } // try
} // doGet
```

A SAX reader (obtained from a pool of readers) reads the content of the XHTML template file and invokes the Hamlet's `getElementReplacement()` callback method when a <REPLACE> tag is encountered. The code in the `getElementReplacement()` callback replaces the placeholder values with dynamically created content based on the value of the ID attribute stored in the `id` parameter.

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

  if (id.equals ("time")) {
    Date now = new Date ();
    String str = dateFormat.format (now);
    return str;
  } // if
  return "?";

} // getElementReplacement
```

For example, to implement a simple digital clock using a Hamlet, see the XHTML template in Listing 2 and the Hamlet Java code in Listing 3:

Listing 2. XHTML template file for a simple digital clock

```
<HTML>
  <HEAD>
    <META HTTP-EQUIV="Refresh" CONTENT="10" />
    <TITLE>Clock</TITLE>
    <LINK REL="stylesheet" TYPE="text/css" HREF="Status.css" />
  </HEAD>
  <BODY>
    <TABLE CLASS="container">
      <TR>
        <TD HEIGHT="30">
          <P CLASS="time">
            <REPLACE ID="time">12:34</REPLACE>
          </P>
        </TD>
      </TR>
    </TABLE>
  </BODY>
</HTML>
```

Listing 3. Hamlet Java code for a simple digital clock

```
package com.ibm.webzec.apps.server;

import java.io.*;
import java.util.*;
import java.text.*;
import javax.servlet.*;
import javax.servlet.http.*;
import com.ibm.hamlet.*;
import com.ibm.webzec.libs.util.*;
import org.xml.sax.*;

public class Clock extends Hamlet {

  private  SimpleDateFormat  dateFormat;

  public void init () throws ServletException {
    try {
      // get properties
      ContextProperties props = ContextProperties.getProperties (this);
      // get formats
      String format = props.getStringProperty ("ShortTimeFormat");
      dateFormat = new SimpleDateFormat (format);
      String timeZone = props.getStringProperty ("TimeZone");
      dateFormat.setTimeZone (Utilities.getTimeZone (timeZone));
    } catch (Exception e) {
      throw new ServletException (e);
    } // try
  } // init

  public synchronized void doGet (HttpServletRequest req, HttpServletResponse res)
    throws ServletException {
    try {
      // serve document
      serveDoc (req, res, "ClockTemplate.html");
    } catch (Exception e) {
      throw new ServletException (e);
    } // try
  } // doGet

  public String getElementReplacement (String id, String name, Attributes atts)
    throws SAXException {
    if (id.equals ("time")) {
      Date now = new Date ();
      String str = dateFormat.format (now);
```

```
      return str;
    } // if
    return "?";
  } // getElementReplacement

} // Clock
```

<REPEAT> and getElementRepeatCount()

With the <REPEAT> tag and the `getElementRepeatCount()` callback, you can repeat sections in an XHTML template file several times. This functionality is useful to create the rows of a table, the selections in a drop-down list, conditional includes, or other repeated items. A Web designer adds the <REPEAT> tag (together with an ID attribute) to those places within the XHTML file that are repeated.

```
<REPEAT ID="rows">
   <TD><P CLASS="row"> 
      <REPLACE ID="Reviewed">Y</REPLACE> </P></TD>
   <TD><P CLASS="row"> 
      <REPLACE ID="Reception">05/24/04 12:47:24</REPLACE> </P></TD>
   <TD><P CLASS="row"> 
      <REPLACE ID="Generation">05/24/04 12:47:16</REPLACE> </P></TD>
   ...
</REPEAT>
```

As I mentioned before, once the Hamlet receives the user request, its `doGet()`/`doPost()` method is invoked. The method passes the name of the XHTML template file to the `serveDoc()` method of the Hamlet framework where its content is read by a SAX reader. If the reader comes across a <REPEAT> tag, it records all following XHTML content. When the reader encounters the corresponding </REPEAT> tag, it stops the recording and invokes the `getElementRepeatCount()` callback. The code in the `getElementRepeatCount()` callback returns the number of repeats N with the help of the ID attribute stored in the `id` parameter. Next, the recorded content plays back *N* times where *N* equals the number of repeats. You can nest <REPEAT> tags.

```
public int getElementRepeatCount (String id, String name, Attributes atts)
  throws SAXException {

  if (id.equals ("rows"))
    return events.size ();
  return 0;

} // getElementRepeatCount
```

For example, to implement a table containing intrusion-detection events using a Hamlet, see the XHTML template in Listing 4 and the Hamlet Java code in Listing 5:

Listing 4. XHTML template file for a table containing intrusion-detection events

```
<!DOCTYPE WebZEC [ <!ENTITY nbsp "Â "> ]>
<HTML>
  <HEAD>
    <TITLE>ListView</TITLE>
    <LINK REL="stylesheet" TYPE="text/css" HREF="View.css" />
  </HEAD>
  <BODY>
    <TABLE CLASS="list" CELLSPACING="0" CELLPADDING="0">
      <TR>
```

```
        <TD></TD>
        <TD><P CLASS="header">Received</P></TD>
        <TD><P CLASS="header">Generated</P></TD>
        <TD><P CLASS="header">Signature</P></TD>
        <TD><P CLASS="header">Count</P></TD>
        <TD><P CLASS="header">Source IP</P></TD>
        <TD><P CLASS="header">SP</P></TD>
        <TD><P CLASS="header">Dest. IP</P></TD>
        <TD><P CLASS="header">DP</P></TD>
        <TD><P CLASS="header">Customer</P></TD>
        <TD><P CLASS="header">Sensor</P></TD>
      </TR>
      <REPEAT ID="rows">
      <TR ID="Color" BGCOLOR="#FFFFFF">
        <TD><P CLASS="row"> 
          <REPLACE ID="Reviewed">Y</REPLACE> </P></TD>
        <TD><P CLASS="row"> 
          <REPLACE ID="Reception">05/24/04 12:47:24</REPLACE> </P></TD>
        <TD><P CLASS="row"> 
          <REPLACE ID="Generation">05/24/04 12:47:16</REPLACE> </P></TD>
        <TD><P CLASS="row"> 
          <A ID="Link"
            HREF="EventView.html?Index={0}" CLASS="dynamic TARGET="APPLICATION">
          <REPLACE ID="Signature">
             RPC - mountd UDP unmount request: Attempted Information Leak
          </REPLACE>
          </A> </P>
        </TD>
        <TD><P CLASS="row"> 
           <REPLACE ID="Count">10</REPLACE> </P></TD>
        <TD><P CLASS="row"> 
           <REPLACE ID="SrcIP">127.0.0.1</REPLACE> </P></TD>
        <TD><P CLASS="row"> 
           <REPLACE ID="SrcPort">80</REPLACE> </P></TD>
        <TD><P CLASS="row"> 
          <REPLACE ID="DstIP">129.173.21.68</REPLACE> </P></TD>
        <TD><P CLASS="row"> 
           <REPLACE ID="DstPort">44</REPLACE>3 </P></TD>
        <TD><P CLASS="row"> 
           <REPLACE ID="Customer">Pawlitzek AG</REPLACE> </P></TD>
        <TD><P CLASS="row"> 
           <REPLACE ID="Sensor">S1</REPLACE> </P></TD>
      </TR>
      </REPEAT>
    </TABLE>
  </BODY>
</HTML>
```

Listing 5. Hamlet Java code for a table containing intrusion-detection events

```
package com.ibm.webzec.apps.server;

import java.awt.*;
import java.io.*;
import java.util.*;
import java.text.*;
import javax.servlet.*;
import javax.servlet.http.*;
import com.ibm.hamlet.*;
import com.ibm.webzec.libs.db.*;
import com.ibm.webzec.libs.util.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;

public class ListViewList extends Hamlet {

  private  int               index;
```

```
private  Vector            events;
private  EventDesc         event;
private  SimpleDateFormat  dateFormat;

public void init () throws ServletException {
  try {
    // get properties
    ContextProperties props = ContextProperties.getProperties (this);
    // get formats
    String format = props.getStringProperty ("DateFormat");
    dateFormat = new SimpleDateFormat (format);
    String timeZone = props.getStringProperty ("TimeZone");
    dateFormat.setTimeZone (Utilities.getTimeZone (timeZone));
  } catch (Exception e) {
    throw new ServletException (e);
  } // try
} // init

public synchronized void doGet (HttpServletRequest req, HttpServletResponse res)
  throws ServletException {
  try {
    HttpSession session = req.getSession (false);
    if (session != null) {
      index = 0;
      events = (Vector) session.getAttribute (WebApp.events);
      // serve document
      serveDoc (req, res, "ListViewListTemplate.html");
    } else {
      res.sendRedirect (res.encodeURL (req.getContextPath () + "/LoginHere.html"));
    } // if
  } catch (Exception e) {
    throw new ServletException (e);
  } // try
} // doGet

public int getElementRepeatCount (String id, String name, Attributes atts)
  throws SAXException {
  if (id.equals ("rows"))
    return events.size ();
  return 0;
} // getElementRepeatCount

public String getElementReplacement (String id, String name, Attributes atts)
  throws SAXException {
  if (id.equals ("Reviewed")) {
    return event.isReviewed () ? "Y" : "-";
  } else if (id.equals ("Reception")) {
    return dateFormat.format (event.getReceptionDate ());
  } else if (id.equals ("Generation")) {
    return dateFormat.format (event.getGenerationDate ());
  } else if (id.equals ("Signature")) {
    return event.getSignature ();
  } else if (id.equals ("Count")) {
    return "" + event.getEventCount ();
  } else if (id.equals ("SrcIP")) {
    return event.getSrcIP ();
  } else if (id.equals ("SrcPort")) {
    return "" + event.getSrcPort ();
  } else if (id.equals ("DstIP")) {
    return event.getDstIP ();
  } else if (id.equals ("DstPort")) {
    return "" + event.getDstPort ();
  } else if (id.equals ("Customer")) {
    return event.getCustomerName ();
  } else if (id.equals ("Sensor")) {
    return event.getSensorName ();
  } // if
```

```
    return "?";
  } // getElementReplacement

  public Attributes getElementAttributes (String id, String name, Attributes atts)
    throws SAXException {
    if (id.equals ("Color")) {
      // get the Event descriptor
      event = (EventDesc) events.elementAt (index);
      Color color = event.getColor ();
      int rgb = color.getRGB ();
      String col = Integer.toHexString (rgb);
      AttributesImpl atts2 = new AttributesImpl (atts);
      atts2.setValue (atts2.getIndex ("BGCOLOR"), "#" + col.substring (2));
      atts = atts2;
    } else if (id.equals ("Link")) {
      String format = atts.getValue ("HREF");
      String tmp = "" + index;
      Object[] arg = { tmp };
      tmp = MessageFormat.format (format, arg);
      AttributesImpl atts2 = new AttributesImpl (atts);
      atts2.setValue (atts2.getIndex ("HREF"), tmp);
      atts = atts2;
      // increment index
      index++;
    } // if
    return atts;
  } // getElementAttributes

} // ListViewList
```

getElementAttributes()

The `getElementAttributes()` callback is the technique to add, remove, or change the attributes of elements in the XHTML code. Mark the elements that must have their attributes processed with an ID attribute. During the parsing of the XHTML template file, the SAX reader calls the `getElementAttributes()` method for each element with an ID attribute. The element's `id` attribute selects the appropriate processing.

In the previous example, the `getElementAttributes()` callback colors each table row according to the severity of an event. That is, fatal and critical events display in red, warning events display in yellow, and harmless events display in green.

```
if (id.equals ("Color")) {
  // get the Event descriptor
  event = (EventDesc) events.elementAt (index);
  Color color = event.getColor ();
  int rgb = color.getRGB ();
  String col = Integer.toHexString (rgb);
  AttributesImpl atts2 = new AttributesImpl (atts);
  atts2.setValue (atts2.getIndex ("BGCOLOR"), "#" + col.substring (2));
  atts = atts2;
} // if
```

To create a link for each row, use the `getElementAttributes()` callback as well. The XHTML file contains a template for the link (`HREF="EventView.html?Index={0}`). The actual value of the event index fills the template.

```
if (id.equals ("Link")) {
  String format = atts.getValue ("HREF");
  String tmp = "" + index;
  Object[] arg = { tmp };
  tmp = MessageFormat.format (format, arg);
  AttributesImpl atts2 = new AttributesImpl (atts);
  atts2.setValue (atts2.getIndex ("HREF"), tmp);
  atts = atts2;
  // increment index
  index++;
} // if
```

<INCLUDE> tag

To insert content from another source into the XHTML code, use the <INCLUDE> tag. With it, you can create headers, footers, or sections to reuse on multiple pages (such as copyright statements).

```
<HTML>
  <HEAD>
    <TITLE>Welcome to WebZEC</TITLE>
    <LINK REL="stylesheet" TYPE="text/css" HREF="View.css" />
  </HEAD>
  <BODY>
    <P CLASS="title">
      Welcome to WebZEC!
    </P>
    ...
    <INCLUDE ID="Copyright" SRC="Copyright.html" />
  </BODY>
</HTML>
```

To indicate the source to include at runtime, you can also use the `getElementAttributes()` callback.

```
public Attributes getElementAttributes (String id, String name, Attributes atts)
  throws SAXException {
  if (id.equals ("Copyright")) {
    AttributesImpl atts2 = new AttributesImpl (atts);
    atts2.setValue (atts2.getIndex ("SRC"), "Copyleft.html");
    atts = atts2;
  } // if
  return atts;
} // getElementAttributes
```

You've now seen the full functionality that the Hamlet framework offers.

Develop Hamlet-based Web apps

To develop a Hamlet-based Web application, follow this simple process:

1. In a (X)HTML editor, a Web designer creates XHTML template files that contain the presentation of a Web application. The designer includes placeholder values (dummy values) for those places that are replaced by dynamic content when the pages are accessed during runtime. In addition, the Web designer writes Cascading Style Sheets (CSS) to store formatting instructions which provide a consistent look and feel for all Web pages.

2. Next, mark all sections within the XHTML template file that you want dynamically created or repeated with <REPLACE> and <REPEAT> tags together with ID attributes. Furthermore, add an ID attribute to elements whose attributes you want to process. Note that the ID attribute is the only link between content and presentation; it separates the two completely.
3. The Java developer implements the three callback methods -- `getElementReplacement()`, `getElementRepeatCount()`, and `getElementAttributes()` -- in a Hamlet to provide the dynamic content. The ID attribute controls the creation of dynamic content in the callback functions.
4. Finally, the developer compiles the Java code, jars the class files and XHTML files, and deploys the Web application.

Intentionally, this procedure is similar to the well-known classic Windows GUI development. Many developers are familiar with this software-creation method that dominated the Windows application-development process before Rapid Application Development (RAD) became popular:

1. In a dialog editor, a GUI designer creates resource files (.rc) that contain the controls of a dialog.
2. She replaces all numerical control IDs created by the dialog editor with C/C++ preprocessor names:
```
#define ID_ABOUT_DIALOG_OK        10000
#define ID_ABOUT_DIALOG_CANCEL    10001
#define ID_ABOUT_DIALOG_HELP      10002
```
3. The C/C++ developer uses the preprocessor names in the code to access the controls in the dialog. In the following example, `GetDlgItem()` returns the handle to the OK button in the dialog. The handle is then used in `setWindowText()` to set the control's text property.
```
setWindowText (GetDlgItem (hwnd, ID_ABOUT_DIALOG_OK), buffer);
```
4. In a last step, the developer compiles the C/C++ code and the resource files (.rc), links them together to form an .exe file, and installs the application.

Developers who are familiar with the classic Windows GUI development process will immediately be able to understand and use the Hamlet framework. Both environments achieve complete separation of content and presentation by using IDs.

Traditional versus rapid application development

Like most other servlet-based content creation frameworks, Hamlets are practical for traditional, Web-based application development. Rapid Application Development (RAD) became popular with the introduction of Visual Basic and Borland Delphi. These IDE tools allow the quick development of standalone applications with integrated GUI builders. For the development of Web-based applications, such tools are not so widespread yet.

Table 2. What appdev form is right?

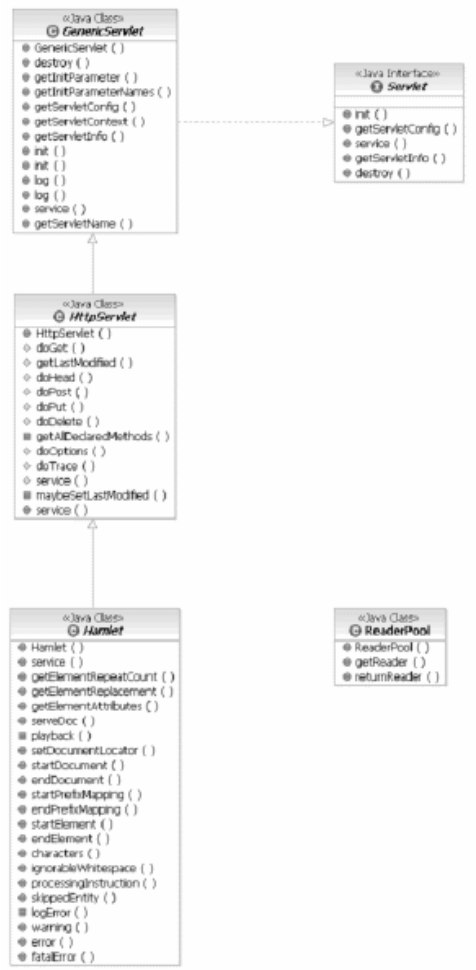
	Traditional application development	Rapid application development
Web-based application development	• Most servlet-based content creation frameworks • Hamlets	• Web forms • JavaServer Faces (JSF)

Standalone application development	• Classic Windows • OS/2 Presentation Manager	• Windows Forms (Visual Basic) • Borland Delphi, C++Builder • Various other RAD tools

Implement Hamlets

Hamlets form a lightweight, servlet-based content creation framework that enforces the complete separation of content and presentation. The implementation consists of two (public) Java classes with less than 500 lines of Java source code (see Figure 1).

Figure 1. UML diagram showing the Hamlet class hierarchy



The SAX library does most of the work (the XHTML parsing). The rest of the framework consists of these parts:

- The `ReaderPool` class provides a mechanism to store SAX readers between requests. Reuse of the readers increases the performance of the framework. The implementation is 66 lines long (including a commented header with 17 lines).
- The `Hamlet` class (an extension of HttpServlet) includes the logic to set up the SAX reader, to record and play back XHTML contents, and to invoke the callback functions. It contains 388 lines of code (including a commented header with 23 lines).

I might have implemented the Hamlet framework in various other ways; the implementation used in this article represents a keep-it-simple approach.

Summary

Servlets are a key component of server-side Java development, offering such benefits as portability, efficiency, safety, extensibility, and flexibility. Unfortunately, servlets do not support or enforce the complete separation of content from presentation. Of the many frameworks devised to provide this support, none were suitable for a Web-based console project due to license, complexity, performance, and security.

The lightweight Hamlets -- they contain just two Java classes with less then 500 lines of Java source code -- are the answer for a small-footprint, servlet-based content creation framework. The Hamlet framework offers the complete separation of content and presentation, doesn't hide the familiar underlying servlet infrastructure, and is easy to use and understand.

The development of Hamlet-based Web application is similar to the development of traditional standalone applications in the classic Windows environment, so Windows programmers can start immediately and produce results quickly.

Resources

- Get an overview of Java servlets in The Java Servlet API White Paper.
- In A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System, explore MVC programming, the application of a three-way factoring with which objects of different classes take over the operations related to the application domain, the display of the application's state, and the user interaction with the model and the view.
- Discover 23 patterns that are an essential resource for anyone developing reusable software designs in *Design Patterns: Elements of Reusable Object-Oriented Software*.
- Check out this White Paper JavaServer Pages Technology for an overview of the JSP technology.
- Get the details on tools and products for building dynamic, Web-based applications in JavaServer Pages White Paper.
- Investigate how to develop servlets in the JavaServer Pages Servlet Developer White Paper.
- Compare CGI, mod_perl, and PHP with the Java Servlet and JSP for creating dynamic content in JavaServer Pages Comparing Methods for Server-Side Dynamic Content White Paper.
- Look at WebMacro, a Java open source template language that can be an alternative to JSP.
- Visit the Enhydra XMLC Project for presentation technology that delivers strict separation of markup and logic in a true object view of dynamic presentations.
- Try the Element Construction Set, a Java API for generating elements for various markup languages.
- Check out SAX, the Simple API for XML.
- Decide which scripting language is for you in Server-side scripting languages (developerWorks, April 2001).
- Take this hands-on tutorial to better understand Java servlets, Introduction to Java Servlet technology (developerWorks, December 2004).
- In Using HttpServlet init method, read up on the use of the HttpServlet init method (developerWorks, August 2001).
- Browse for books on these and other technical topics.
- Visit the developerWorks Web Architecture zone for hundreds of articles and tutorials about various Web-based solutions.
- Get involved in the developerWorks community -- participate in developerWorks blogs.

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 on Security Information Management (SIM) solutions for the GSAL at the IBM Zurich Research Laboratory in Switzerland. Before coming to IBM, he worked in California for Hewlett-Packard, WindRiver Systems, and Borland International.