

A Filter-Based Context-Aware Cross-Site Logging Framework for MVC Web Applications

Adrian E. Ridner, Mei-Ling L. Liu
California Polytechnic State University, San Luis Obispo
Computer Science Department
California, U.S.A.
mliu@csc.calpoly.edu

Abstract

This paper describes the concepts and development of a framework that allows information to be collected about a user's activities during a web session that may span multiple sites. The framework is integrated with the Model View Controller architecture for web development and can be deployed to log information about sessions, clicks, and impressions. The framework is designed for ease of integration with the deployment environment.

Keywords: web application logging, model view controller framework, web development.

1. Introduction

For website providers, it is highly desirable to obtain statistics on the activities of a visitor during a session. Such statistics can be used to maximize the effectiveness of a website. Currently, there is an abundance of tools such as *rcounter*[12], *statCounter*[13], *webStats*[14] for measuring traffic and monitoring site usage. These tools, based on counters and server access files, provide information such as the amount of visitors to a site or the characteristics of a visitor (in terms of the screen resolution and the browser software that he or she is using.) This type of information, while useful, is limited in scope to a specific page or site rather than an entire user session that typically spans multiple pages and often multiple sites.

This paper describes a logging framework for web applications that are developed with Model-View-Controller (MVC) frameworks[1] running on Java 2 Enterprise Edition (J2EE) Web Application Servers[4]. The framework is designed to allow the logging of

events that occur across multiple sites. The events logged

are context-aware in the sense that they are customizable to the context of a site. Furthermore, the framework can be deployed without requiring any change in the source code of the web application being logged.

1.1. MVC Web Application Frameworks

MVC is a well-known architectural design pattern for interactive applications. The MVC model can be adapted to J2EE web applications. In such applications, there are three component types: *model* components are implemented using java beans, *view* components using java server pages (JSPs), and *controller* components using one or more *servlets* [6]. Each component type is used to its strengths: the request controllers handle the request dispatching, the view components the presentation logic, while the model components encapsulate the application logic.

A number of tools, termed application frameworks, are available to provide a framework for building Java web applications based on the MVC model. Among such tools, the most widely adopted is the open source *Apache Struts*[2, 7]. Using the *Struts* framework, each web application is associated with a deployment descriptor, for the initialization of resources such as *servlets* and *taglibs* (tag libraries) when the application is deployed. The deployment descriptor is an Extensible Markup Language (XML) document named *web.xml*. Using *Struts*, the steps for building a web application are as follows:

1. Implement data entry forms as JSP files.
2. Implement one or more *ActionForm* descendents to buffer data between JSPs and Actions (servlets).
3. Create an XML document that defines the validation rules for the application.

4. Implement one or more *Action* descendents to respond to form submissions.
5. Create an XML file named *struts-config.xml* to associate forms with actions.
6. Create or update *web.xml* to reference *ActionServlet*, *taglibs* used by *Struts*.

The success of *Struts* has inspired other application frameworks, including *Maverick*[10], and *WebWork* [11]. All three frameworks support a facility known as *filters*[9], to be described, which can be easily added to the configuration of an application.

1.2. J2EE Filters

In the context of a J2EE Web Application, a filter is a software component that dynamically intercepts requests and responses to transform or use the information contained in the requests or responses. A web application can be configured such that whenever a request for a specific resource (as identified by the URL specified with the request) is received by the HTTP server, the code in one or more filters is automatically executed.

When a filter intercepts a request, it has access to Java objects that encapsulate the HTTP request and response. When an HTTP request calls for a resource that has been bound with a filter, the filter, a Java object, is executed. A filter can examine and modify the response headers and data, block the request, throw an exception, or invoke the execution of another filter. Filters can be employed effectively for logging, debugging, and for implementing functionalities for encryption, and data compression.

J2EE filters play a significant role in our logging framework.

2. Logging Framework Design

Our logging framework consists of an application program interface (API) and a set of J2EE filters.

The API is composed of an *EventLogger* interface and a set of classes, termed *loggers*, that implement specific logging functionalities for various types events. By design, the logger classes are extendable for customized logging. In addition, the framework provides a set of filters, each of which is aware of the specific logger that it is meant to create and is capable of requesting a *LoggerManager* (a factory) for creating the correct type of logger. Once a logger has been created, the logger calls the *log* method in the logger for writing the appropriate log records to a file.

2.1. The EventLogger Interface

The *EventLogger* interface encapsulates the common attributes of an event to be logged. The interface specifies a *log* method that accepts an *Object* as a parameter, to be cast to a type of event object as appropriate to the logger. Each event object contains a method to update a file handler, a Java Handler object responsible for writing the logged data to the files system. The *EventLogger* interface also specifies a method to initialize any database-related objects, if they are required. Last, the interface defines two methods to obtain the path to the application's directory and its logging directory.

2.2. BaseEventLogger

The *BaseEventLogger* is an abstract class that implements the *EventLogger* interface. Its purpose is to provide utility methods common to all loggers.

2.3. BaseClickLogger

The most general type of event in a web application is a click (the selection of a link, for example). A *BaseClickLogger* derives from the *BaseEventLogger* and it is not context-aware. It contains functionality required by classes that will need to record any type of "click", including the basic implementation of the *updateFileHandlerIfNeeded* and *log* methods from the *EventLogger*.

2.4. BaseSearchLogger

Many websites provide some form of a search engine to allow their visitors to perform a search. The *BaseSearchLoggerEvent* abstract class extends the *BaseEventLogger* and allows each search to be logged. This class is to be extended by customized derived classes that are context-aware: That is, the derived classes are implemented for a specific web page, and contains the logic for what specific type of search events is to be logged.

2.5. ClickToFormLogger

This class encapsulates the logging functionality required when a user clicks on (selects a link that forwards the visitor to) an external website, called a *form*. The *ClickToFormLogger* extends the *BaseEventLogger*. Among the methods this class provides is one named *assignGuid()*. This method generates a global unique identifier (*guid*) to the click,

and it includes the *guid* in the event record written to the log. When the user is forwarded to the external website, the *guid* is passed as an HTTP GET parameter. The *guid* serves as a tag that allows for cross-site events that belong to the same session to be correlated.

2.6. ImpressionsLogger

In web development parlance, an impression is an event that occurs when some item meant to entice attention is displayed (or impressed) to a visitor in a web page. For example, a graphic display may appear at a strategic spot on a particular page to invite a user to explore a specific group of merchandises, and the efficiency of such impressions is of interest to website providers. The *ImpressionsLogger* class extends the *BaseEventLogger* and can be customized to monitor the events related to the impressions for a specific web application.

2.7. ClickLogger

The *ClickLogger* class derives almost all its functionality from the *BaseClickLogger* and are customized to log the clicks for a specific web application.

2.8. SearchLogger

The *SearchLogger* derives its functionality from the *BaseSearchLogger* and are customized to log the searches for a specific web application.

2.9. ClickLoggerFilter

This class implements the Java *Filter* interface. This is a class that encapsulates the filter activated when a click event takes place.

2.10. SearchLoggerFilter

This class also implements the Java *Filter* interface and represents the filter activated when a search event occurs.

2.11. The LoggerManager

The *LoggerManager* is a class responsible for implementing the design pattern that ensures that each logger object is a singleton; that is, it ensures that there can only be a single instance of each type of logger for a particular web application. The *LoggerManager* uses reflection to achieve this goal. When a filter requires a logger, the corresponding *Class* object (such as

ClickLogger.class) is passed to the *LoggerManager.getInstance(Class c)* method. The *LoggerManager* has a local Map where it maintains all its singleton instances. It first attempts to look up an existing instance of the class by using the *Class* object passed in as the key to this map. If there is a corresponding value, it will be an existing instance of the required logger. If the map doesn't contain a matching value, the *Class* object is used to create a new instance. The new instance is then inserted into the map.

2.12. LoggerRotator

The *LoggerRotator* class is implemented using the strategy design pattern. The strategy design pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Hence, the design pattern allows an algorithm to vary according to its usage[1]. In *LoggerRotator*, this pattern is used to provide a flexible granularity for the rotation of the log files (see "Log file granularity" in the next section.) The *LoggerRotator* class has a *updateFileHandlerIfNeeded()* method which that encapsulates the strategy for saving log files. Namely, the current format for the name and location of all log files is defined here and can be easily changed without affecting any of the calling code.

2.13. LogIterator

Similar to the *LogRotator*, this class employs the strategy design pattern to encapsulate the strategy for how to iterate through the log files created by this. This class implements the Java *Iterator* interface.

3. Logging Framework Implementation

To facilitate the installation of our logging framework, an Apache *Ant* [15] task was created for compiling all the source code and packaging them in a *jar* file.

3.1. XML Configuration for Filters

For the deployment of our framework, a web application must be configured with the necessary filters. The XML code to configure a filter in a web application involves a simple step of modifying the *WEB-INF/web.xml* file to contain a mapping for the *ClickLoggerFilter*, as illustrated below:

```
<filter>
  <filter-name>
    ClickLoggerFilter
```

```

</filter-name>
<filter-class>
  com.remilon.filters.ClickLoggerFilter
</filter-class>
</filter>
<filter-mapping>
  <filter-name>
    ClickLoggerFilter
  </filter-name>
  <url-pattern>
    /exec/*
  </url-pattern>
</filter-mapping>

```

The XML code above specifies that every request to a webpage whose URL contains the substring “/exec” is to be first “filtered” by our *ClickLoggerFilter*. This example assumes that all significant pages in the web application have a URL that contains the path /exec/, as in <http://somesite.com/exec/logon.jsp> or <http://othersite.com/blah/hi/exec/other.jsp>. Most web applications already employ such a URL hierarchy [4].

Each logger must be mapped to at least one filter. Thus, if a search logger is to be activated, then the configuration file will need to be modified to include the following code for the *SearchLoggerFilter*, as follows:

```

<filter>
  <filter-name>
    SearchLoggerFilter
  </filter-name>
  <filter-class>
    com.remilon.filters.SearchLoggerFilter
  </filter-class>
</filter>
<filter-mapping>
  <filter-name>
    SearchLoggerFilter
  </filter-name>
  <url-pattern>
    /search/*
  </url-pattern>
</filter-mapping>

```

There are two things of note here. First, since both filter definitions appear in the configuration file for one application, both filter mappings will be in effect for each HTTP request to this application. For example, if the URL is <http://something.com/blah/exec/search?id=1>, then both the *ClickLoggerFilter* filter and the *SearchLoggerFilter* filter object will be executed in the order that they are defined in the web.xml file [5]: The *ClickLoggerFilter* will detect the mapping /exec in the URL and run, while

the *SearchLoggerFilter* will detect the /search substring in the URL and run.

Second, the added XML lines are standard J2EE filter mappings and will work on any web application server that is compliant with Servlet version 2.2, such as Apache Tomcat[7], WebLogic[8], and Resin[3,8].

3.2. Detecting the Start and End of a Session

Since our framework is to allow the logging of events throughout a session, it is important to be able to identify the start and the end of a session. At the start of a session, a unique identifier -- a global user ID (*guid*)[16] -- is generated to tag the events generated throughout the session. Determining the beginning of a session is performed in the *BaseClickLoggerFilter* by querying the *HttpRequest* object, as follows:

```

.....
httpReq = ((HttpServletRequest)request);
session = httpReq.getSession();
if (session.isNew( ))
{
  this.logNewSession(httpReq, session);
}
...

```

Determining the end of a session was a more complicated matter. A session may end when a user abruptly leaves the site, closes the browser window, loses his/her internet connection, or takes many other actions without signing off. The typical solution is to use a time-out concept so that after a threshold of inactivity a session is considered closed [5]. For the purpose of this framework, however, the logged information can be analyzed to determine the last time that a user (identified by a *guid*) performed an action on any of the sites where logging is enabled.

3.3. Tracking cross-site activities

The primary novelty of this framework is its capability to track cross-site activities. The implementation of this capability involves the use of a filter that is activated whenever an HTTP request is received: the filter affixes to the request’s URL a *guid*. Thus, for example, if a user clicks on a link such as <http://gotootersite.com/exec/browse.jsp> the filter will intercept that request and change the link to “attach” a random *guid* to the URL, resulting in the submission of the following link to the server:

<http://gotoothersite.com/exec/browse.jsp?guid=z8r4u7dn459skmdn45tjdfis93k54mf0s8>.

Two other conditions must be satisfied:

- The web application on *gotoothersite.com* must be running the same logging framework. The logging framework will detect that the *guid* parameter was included in the HTTP request query string and will associate the *guid* with the identifier that it assigns for the new session (on *gotoothersite.com*). The association must be logged on *gotoothersite.com*.
- When the intercepting filter runs it must also log an association between its local id for the user and the *guid* sent to *gotoothersite.com*.

These two conditions allow the data to be correctly correlated in later analysis.

The critical lines that perform the actions to satisfy these conditions are bolded in the code snippet show below:

```
{
    ...
    String guid = this.assignGuid( );
    String msg = QUOTE + guid + QUOTE +
                SEP + sessionSiteId + SEP +
                linkIdStr + SEP + dbSessionId
                + SEP;
    msg += this.quoteIfNotNull
           (httpReq.getHeader("Referer"));
    msg += QUOTE + new
           String(httpReq.getRequestURL( ))
           + QUOTE + SEP;
    msg += this.quoteIfNotNull
           (httpReq.getQueryString( ));
    java.sql.Timestamp now = new
           java.sql.Timestamp(
           new Date( ).getTime( ));
    msg += now.toString( );
    // log sql statement
    ClickLogRecord logRecord = new
    ClickLogRecord(Level.FINEST, msg);
    clicktoformLogger.log(logRecord);
    log.finer("Logging click to form end");
    httpReq.setAttribute("guid", guid);
    return httpReq;
}

private String assignGuid( )
{
    String guid = new
    RandomGuid(true).toString( );
    return guid;
}
```

3.4. Log file granularity

Using our framework, data are collected throughout the lifetime of a web application. The volume of the data will grow unboundedly unless the files for recording the data is continuously reused. For our implementation, the log files are kept on an hour-based format: every hour a file that is being used to record a given log event is reused. The log files are organized in as follows:

`/base/logger/dir/YEAR/MONTH/DAY/HOUR`

For example: The file `/base/logger/dir/2004/01/03/22` contains the data collected from hour 22 (10PM to 11PM) January 3 of 2004.

4. Deployment Results

A prototype of our logging framework was deployed on six interconnected sites to assess (i) the ease of deployment, and (ii) the performance and scalability of the framework. The framework was deployed with six established web applications running on *Resin* application servers. The applications represent a set of websites for vocational schools. Combined, the network of sites services over 40,000 sessions daily.

4.1. Server configuration

The web applications ran on separate server hosts. Each application used the Struts Framework and had a sizeable code base. Each server machine had a Dual PIII 1.8 GHZ processor, 2 GB of RAM and 200 GB of storage.

4.2. Installation requirements

The framework was installed on each server host. Each web application has a *lib* directory where *jar* files are kept. The installation process is summarized below:

- Run an *Ant* task to build all the logging framework source into one *jar* file
- Copy the *jar* file (*remilon-logging.jar*) to the *webapp/lib* directory referenced above
- Modify the *WEB-INF/web.xml* file to contain a mapping for the *ClickLoggerFilter*
- Restart the web application server

Optionally, the user may write his/her own filter classes and extend the framework to have additional logger classes. Even so, the installation of the *jar* file and a few lines of XML in the configuration file were sufficient to automatically activate our framework.

For our testing, the logging framework ran for approximately three days without causing significant problems. There were bug fixes as the data volume increased, entailing some synchronization issues and necessitating a few buffering improvements, but for the most part the prototype functioned without impacting the normal operation of the websites, and data was collected as expected. The performance impact was not noticeable.

5. Conclusions

This work presented a filter-based, context-aware, cross-site logging framework that can be integrated with web applications built on a MVC application framework. Our logging framework enables logging of events with negligible performance impact and without requiring any modification of the web application source code. The framework is based on an API that can be extended for customization and for future work.

6. Future work

The implementation of the framework described in this paper is ongoing. Issues that are being explored include employing filtering rules to selectively log by activity or by user.

References

1. Gamma, E. et al. *Design Patterns*. Addison-Wesley. 1995.
2. Goodwill, J. and Hightower, R. *Professional Jakarta Struts*. John Wiley & Sons. 2003.
3. Hightower, R, and Gradecki, J. *Mastering Resin*. Wiley Publishing. 2003.
4. Johnson, R. *J2EE Design and Development*. Wiley Publishing. 2003.
5. R. Eckstein, *Java Enterprise Best Practices*, O'Reilly & Associates, 2002.
6. Sun Microsystems, *JSP and Servlet Specification*, <http://java.sun.com/products/>, 2003
7. The Apache Software Foundation, *Struts*, <http://jakarta.apache.org/struts/>, 2003.
8. BEA Systems, "*WebLogic Application Server*", <http://www.bea.com/>, 2003.
9. Sun Microsystems, *Java Filter Specification*, <http://java.sun.com/products/servlet/Filters.html>, 2003.
10. sourceforge.net, *Maverick MVC Framework*, <http://mav.sourceforge.net/>, 2003.
11. WebWork, *WebWork MVC Framework*, <https://webwork.dev.java.net/>, 2003.
12. NooLab Corporation, *Rcounter*, <http://rcounter.noonet.ru/>, 2004.
13. StatCounter, *StatCounter*, <http://www.statcounter.com/>, 2004.
14. NetLogics, *WebStats*, <http://webstats.netlogics.nl/default.asp>, 2004.
15. Apache Foundation, *The Apache Ant Project*, <http://ant.apache.org/>, 2004.
16. M. Mnich, RandomGUID, <http://www.javaexchange.com/aboutRandomGUID.html>, 2002.