Track wireless sessions with J2ME/ MIDP

Learn three ways to maintain client state information in mobile commerce applications

Summary
Every e-commerce application must support session tracking. Unfortunately, MIDP (Mobile Information Device Profile), a J2ME (Java 2 Platform, Micro Edition) technology, supports only the standard HTTP protocol, which is stateless. In this article, Michael Juntao Yuan and Ju Long explore ways to add session support into the current MIDP network API framework. They discuss the implementations, usages, and relative merits of three approaches: using cookies, rewriting URLs, and embedding session information in XML documents. (*3,000 words;* **April 26, 2002**)

By Michael Juntao Yuan and Ju Long

Client state information proves vital for e-commerce applications that employ user authentication and transactions. In those applications, the server must individually respond to each user according to her previous actions. For example, when a user places an item into a shopping cart, the server must remember and associate that item with that particular user. When the same user selects Checkout later, the server must respond with a list of the merchandise she previously selected.

However, the most widely used Internet e-commerce protocol, HTTP, is stateless. Each HTTP connection is independent and knows nothing about other connections. Standard HTTP connections do not remember client state information. Most developers solve this problem by requiring the client to embed some unique session identification information (usually a session ID string) in each HTTP connection. E-commerce Web servers can then organize individual HTTP connections into sessions according to the session identification. It usually works like so:

1. When a new session's first connection is made, the server generates a new session ID and sends it back to the client

2. The client stores that session ID and attaches it to every subsequent HTTP connection so the server knows all connections belong to the same session

That technique requires cooperation between the client and server software. The client and server must exchange session identification information with a previously agreed upon format. The industry

has developed two de facto standards for exchanging such session information.

One approach transmits small pieces of text, called *cookies*, through HTTP connection headers. The other approach attaches a session ID string to the end of each request URL, a technique also known as *URL rewriting*.

Most Web browsers and desktop HTTP applications support both session-tracking methods. However, the HTTP session support in wireless Java platforms is far from smooth. In the main Java platform designed for cell phones and low-powered PDAs—the Mobile Information Device Profile (MIDP), a Java 2 Platform, Micro Edition (J2ME)-based technology—the HttpConnection object supports neither cookie nor URL rewriting out of the box. Considering the importance of session tracking in e-commerce applications, if we want J2ME/MIDP to be a serious mobile commerce platform, we must equip it with session-aware HTTP connections.

In this article, we discuss how to implement session tracking in MIDP applications with both cookies and URL rewriting. We will also discuss a new way to track sessions by enveloping session information in XML documents. The XML method is unique to wireless applications. (**Note:** We assume you have basic knowledge of MIDP programming. If you need a refresher, please refer to Resources.)

As we mentioned earlier, session tracking requires a joint effort from both the client and the server. In this article, we give examples in the context of Java application servers, but you can easily apply the same techniques to other servers.

Our example is a simple Web visit counter. It uses HTTP sessions to track each user and reports how many times the user connects to the URL. We give framework implementations of this simple Web counter using the above three session-tracking methods. We created a test MIDlet and a test JSP (JavaServer Page) for each method. You can download Java and JSP source code that accompany this article from Resources. Figure 1 shows the sample application's MIDlets."

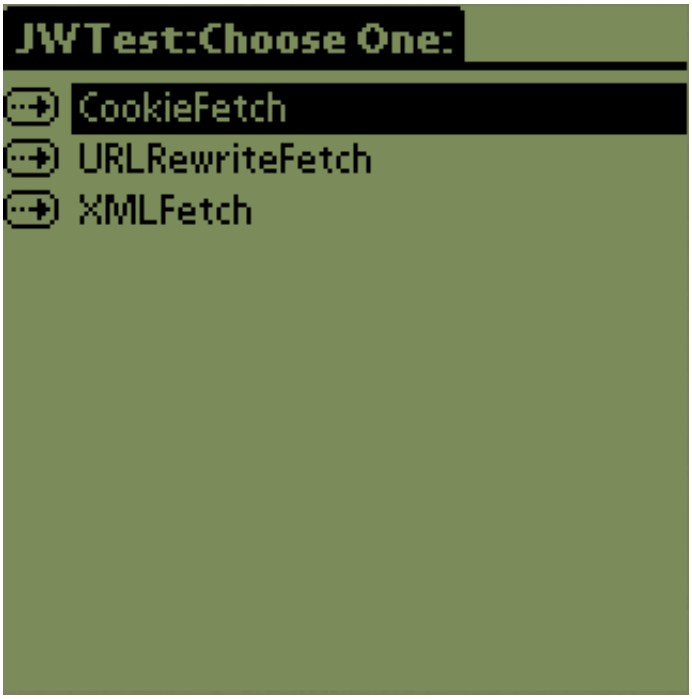

Figure 1. Example application's start screen

Cookies

Cookies are pieces of NAME=VALUE format text embedded in HTTP headers. Netscape originally proposed the cookie concept as an "HTTP State Management Mechanism;" the specification was published in a request for comments, RFC 2109, in 1997. (**Note:** RFC 2695 defines the current cookie specification.) Since cookies reside in HTTP headers, they are transparent to applications and users, and thus the most widely used HTTP session-tracking technique.

The server assigns new cookies to a client through the HTTP header set-cookie. One HTTP connection can have multiple set-cookie headers and hence allows the server to set multiple cookies simultaneously. The set-cookie header takes the format:

```
set-cookie: NAME=VALUE; expires=DATE; path=PATH; domain=DOMAIN_NAME; secure
```

The first NAME=VALUE is the cookie itself and is required. Each cookie can have many optionally specified attributes, such as expiration time, domain, and path. The domain attribute states the cookie's valid domains. It proves especially important since it protects user privacy and reduces the chances for conflicting cookies. For example, a cookie set by yahoo.com is not supposed to be sent out when the user visits amazon.com later. If no domain attribute is specified, a cookie's domain defaults to the host name of the server that sets it.

The client program sends out the valid cookies to each URL it connects to. The client sends cookies back to the server in the HTTP header named cookies:

```
cookie: NAME1=VALUE1; NAME1=VALUE2; ...
```

A client can send multiple cookies in one connection header by delimiting them using semicolons.

Server side

All major Web server vendors support cookie-based HTTP sessions. In this article, we discuss examples in the Java Web server context.

The HttpSession interface in any Java 2 Platform, Enterprise Edition (J2EE)-compatible Java servlet/ JSP container knows how to handle sessions through cookies. To use cookies in our example, we must first ensure that cookie support is not disabled in the configure files. Our example JSP, CookieSession.jsp, uses cookies to track sessions and update visit counters.

As you will see later, an HttpSession object can track sessions using both cookies and URL rewriting, but by default, it uses cookies. So, in our example, if the client's incoming connection sends a valid cookie, the code below returns an existing session object:

```
HttpSession sess = request.getSession(true);
```

Otherwise, the server creates a new session object and sends a new cookie to the client.

The server invalidates sessions after a period of inactivity. That expiration time can be set by server administrators in configure files or by servlets at runtime through the sess.setMaxInactiveInterval() method. We can then associate server-side objects containing client state information with the HttpSession object:

```
sess.setAttribute("Count", count);
```

We can retrieve the above session state information count object later as long as the session remains valid:

```
String count = (String) sess.getAttribute("Count");
```

If no object associates with attribute Count (for example, at the session's beginning), the above statement returns a null value, and we need to start a new counter:

```
if ( count == null ) {
  count = "0";
}
```

MIDP cookie support framework

Supporting cookies on the server side is easy; supporting them in MIDP applications proves more challenging.

We try to make cookie support as transparent as possible; so we borrow a technique from Sun's Smart Ticket J2ME/J2EE demo implementation to design cookie-aware connection objects. RMSCookieConnector is a decorator class for the standard MIDP Connector class. It maintains a MIDP record management system (RMS) record store to store all cookies. When RMSCookieConnector.open() is called, that method completes the following steps:

1. It calls Connector.open() to create a standard MIDP HttpConnection object:

   ```
   HttpConnection c = (HttpConnection) Connector.open(url);
   ```

2. It iterates through the record-store records and fetches the valid cookies. We discuss this step's details in the next section.

3. It assembles the valid cookies into a semicolon-delimited string and sets the string into the HttpConnection object's cookie header:

   ```
   c.setRequestProperty( "cookie", cookieStr );
   ```

4. RMSCookieConnector.open() then wraps the above standard HttpConnection object with a cookie-aware HttpConnection type object implemented by class HttpRMSCookieConnection and returns the HttpRMSCookieConnection instance:

   ```
   HttpConnection sc = new HttpRMSCookieConnection(c);
   return sc;
   ```

The HttpRMSCookieConnection class implements the HttpConnection interface by acting as a decorator for a standard MIDP HttpConnection implementation object wrapped inside a HttpRMSCookieConnection object. Most methods required by the HttpConnection interface in the decorator class pass directly to the wrapped object. But HttpRMSCookieConnection overrides the openInputStream() and openDataInputStream() methods to process the set-cookie fields before returning the stream object. The cookie is simply the string before the first semicolon in set-cookie headers. The code snippet below shows how to retrieve cookies:

```
int k = 0;
while (c.getHeaderFieldKey(k) != null) {
  String key = c.getHeaderFieldKey(k);
```

```
  String value = c.getHeaderField(k);
  if (key.equals("set-cookie")) {
    // Parse the header and get the cookie.
    int j = value.indexOf(";");
    String cValue = value.substring(0, j);

    // Write the cookie into the cookie store.
    // ... ...
  }
  k++;
}
```

This framework hides the cookie-handling process from the application developers. To get a new cookie-aware HttpConnection type object, the application just needs to call the following:

```
HttpConnection conn = (HttpConnection) RMSCookieConnector.open( url );
```

Then we can open input/output streams to communicate with the server. For more details, please refer to the source code of MIDlet CookieFetch, which in turn calls functions from the Utils helper class.

Store cookies in an RMS record store

The RMSCookieConnector class stores cookies in an RMS record store. An RMS record store is accessed by its name, rather than reference, and can persist between soft resets, or reboots. That ensures an old session remains valid when a user later returns to an application after previously quitting it. Considering most people use their mobile information devices for many different tasks simultaneously, cookie persistence is a desired feature.

Remember, RFC 2109 allows us to separate cookies for different sites by examining the domain attribute associated with each cookie. However, implementing the full specifications in RFC 2109 requires rather complex string parsing and proves quite expensive in terms of memory footprint and CPU usage. Instead of being fully specification-compliant, we decided to use a simpler approach: We simply discard all the attributes information accompanying the set-cookie header and associate each cookie with the real host it comes from. We derive the host name from a call to HttpConnection.getHost(). When we connect to that host again, only cookies from the same host will be used.

We store these cookies and their host names in an RMS record store. An RMS record store only has a one-dimensional structure with sequentially ordered data fields. Starting from field number zero, we store cookies in even-numbered fields and associated host names in the odd numbered fields that directly follow each cookie field. The cookie storage and retrieval code is illustrated below:

```
// Get cookies from the connection and store them with host names.
static void getCookie(HttpConnection c) throws IOException {
  RecordStore rs = RecordStore.openRecordStore(cookieStoreName, true);

  // "While" loop to iterate through headers and get cookies
  // in to cValue strings.
  /* Start loop. */
    // Write the cookie into the cookie store.
    int newID = rs.addRecord(cValue.getBytes(), 0, cValue.length());
    // We set the domain default to the current server.
    String dValue = c.getHost();
```

```
   if ( dValue == null ) {
     // If there is no valid domain,
     // we do not keep the cookie.
       rs.deleteRecord(newID);
    } else {
     // All upper case for easy comparison in the future.
     dValue = dValue.toUpperCase();
     // Write the domain into the cookie store.
     rs.addRecord(dValue.getBytes(), 0, dValue.length());
    }
  /* End loop. */

  rs.closeRecordStore();
  return;
}

// Fetch cookies from record store and set into the connection header.
static void addCookie(HttpConnection c, String url) throws Exception {
  String domain;
  // Chunk of code to parse domain from input url.

  StringBuffer buff = new StringBuffer();;
  RecordStore rs = RecordStore.openRecordStore(cookieStoreName, true);
  RecordEnumeration re = rs.enumerateRecords(null, null, false);
  String cookie = "", cookieDomain = "";
  // Iterate through the cookie record store and find cookies
  // with domain matching the current URL.
  //
  // isCookie is used to tell whether the current record is
  // a cookie or an associated domain.
  boolean isCookie = true;
  while ( re.hasNextElement() ) {
    if ( isCookie ) {
      cookie = new String(re.nextRecord());
    } else {
      cookieDomain = new String(re.nextRecord());
      // Cookies are valid for sub-domains.
      if ( domain.endsWith( cookieDomain ) ) {
        buff.append( cookie );
        buff.append("; ");
      }
    }
    isCookie = !isCookie;
  }
  rs.closeRecordStore();

  // If we do have cookies to send, set the composed string into
  // "cookie" header.
  String cookieStr = buff.toString();
  if ( cookieStr == null || cookieStr.equals("") ) {
    // Ignore.
  } else {
    c.setRequestProperty( "cookie", cookieStr );
```

```
  }
  return;
}
```

Following the examples, you can associate each cookie with more properties, such as URL path and expiration time, in real-world applications when such needs arise.

Run the example

Please refer to "Deploy the Example MIDP Application on Palm OS Devices," a sidebar from our previous *JavaWorld* article, for more information on how to build the example programs from the sample code accompanying this article.

Once you set up the application server and MIDP VM properly, you can easily run the example program. Figure 2 shows the initial screen of the MIDlet CookieFetch.

Figure 2. Input a URL for cookie-based session tracking

Put in our server program's URL and press the Fetch button. A test server URL is the above-mentioned JSP counter, CookieSession.jsp. The server returns the number of times you visited the URL during this current session, as Figure 3 illustrates.

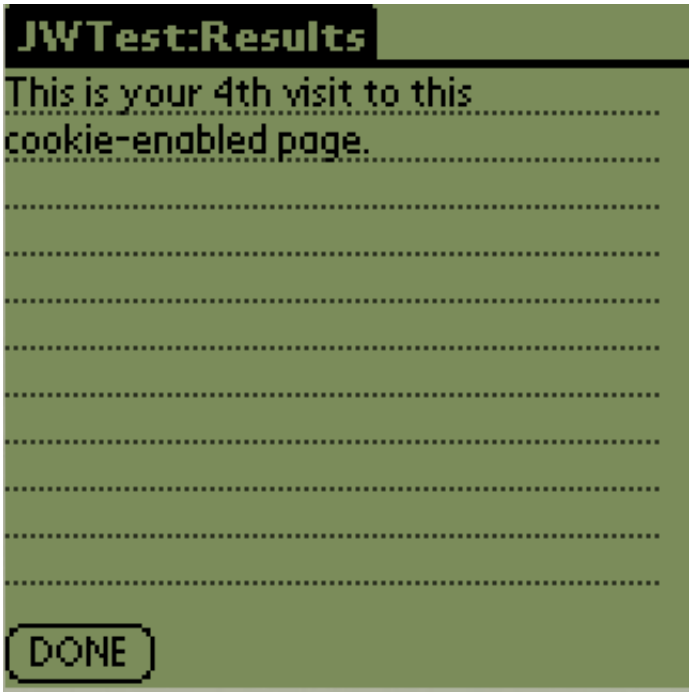

Figure 3. The server page tracks the number of visits

For a lightweight general-purpose library devoted to cookie handling on the device, read "A Recipe for Cookie Management," Sonal Bansal (*JavaWorld*, April 2002).

URL rewriting

Cookies prove easy to use and provide a standard way for session tracking. However, due to some previous cookie abuses, most perceive cookies as unsafe for privacy. As a result, some people hesitate to use cookie-enabled applications. Privacy proves a specific concern for handheld devices because many people use PDAs to manage sensitive personal and financial information. To address this concern, most browsers and servers now have options to turn off cookie support completely.

For cookie-disabled applications, we must use alternative methods to track sessions. One standard failover method for cookies is URL rewriting, which most Web servers support, including J2EE servlet containers. As its name suggests, the URL-rewriting technique embeds session identification information in custom-formatted URLs.

Server side

In the URL-rewriting scheme, when the server encounters a connection that does not belong to any existing sessions, it establishes a new session and generates a unique string that attaches to the end of all URLs associated with subsequent connection requests. To avoid clashes with legitimate URL paths and GET parameters, the attached identification string takes a format resembling the following:

```
;jsessionid=F953068341A94947242185974585D007
```

The server detects all URLs with such attachments and groups them into appropriate sessions for future requests.

In a Java servlet container, to rewrite a URL, you pass the desired URL to the HttpSession.encodeURL() method. As you learned earlier, an HttpSession object can maintain sessions through cookies. It can also support URL rewriting with its internal session identification information. If we pass an empty string to HttpSession.encodeURL(), it returns the URL attachment associated with the current session. Then, the server sends the URL attachment back to the MIDP client through a custom HTTP header. We use the HTTP header url-attmt to pass the URL attachment. The snippet below from URLRewriteSession.jsp illustrates the server actions we discuss above:

```
// Get an existing session or start a new session.
HttpSession sess = request.getSession(true);

// Rewrite an empty URL to get the URL attachment.
String URLAttachment = response.encodeURL( "" );
// Set the response header "url-attmt"
// so that the client can know the URLAttachment.
response.addHeader("url-attmt", URLAttachment);
```

Client side

When the client receives the URL attachment from the url-attmt header, it must store the attachment for future use. The client stores the URL attachment as a static variable in the URLRewriteConnector class. URLRewriteConnector's structure resembles the RMSCookieConnector class. If you need stronger persistence, you can put the URL attachment in an RMS record store and even associate it with a host name as we did for cookies. The code to retrieve a URL attachment from a server response works the same way as the code in RMSCookieConnector:

```
static void getURLAttachment(HttpConnection c) throws IOException {
  // Iterate through headers and get the first "url-attmt" value.
  int k = 0;
  while (c.getHeaderFieldKey(k) != null) {
    String key = c.getHeaderFieldKey(k);
    String value = c.getHeaderField(k);
    if (key.equals("url-attmt")) {
      URLAttachment = value;
      break;
    }
    k++;
  }
}
```

With a valid URL attachment, URLRewriteConnector can open new session-aware connections. URLRewriteConnector.open() rewrites the desired URL and then passes it to the Connector.open() method to get a new connection object. URLRewriteConnector then returns a wrapper object HttpURLRewriteConnection around the new connection object. HttpURLRewriteConnection implements the HttpConnection interface through the Decorator pattern. The following code segment illustrates the above steps in the URLRewriteConnector.open() method:

```
public static HttpConnection open(String url) throws IOException {
  if ( URLAttachment == null ) {
   // Do nothing.
  } else {
```

```
    url = url + URLAttachment;
  }
  HttpConnection c = (HttpConnection) Connector.open(url);
  HttpConnection sc = new HttpURLRewriteConnection(c);
  return sc;
}
```

Run the example

Although tracking sessions using the URL-rewriting technique differs from the cookie technique, their implementations resemble each other. Similar to the cookie approach, with the URL-rewriting approach, we buried the real work in decorator classes to provide transparent session-aware HTTP network connections to applications. Therefore, it should not surprise you that MIDlet URLRewriteFetch resembles MIDlet CookieFetch. The only difference: URLRewriteFetch uses URLRewriteConnector to open new connections, while CookieFetch uses RMSCookieConnector.

Figures 4 and 5 demonstrate the URL-rewriting technique in MIDlet URLRewriteFetch and JSP URLRewriteSession.jsp.

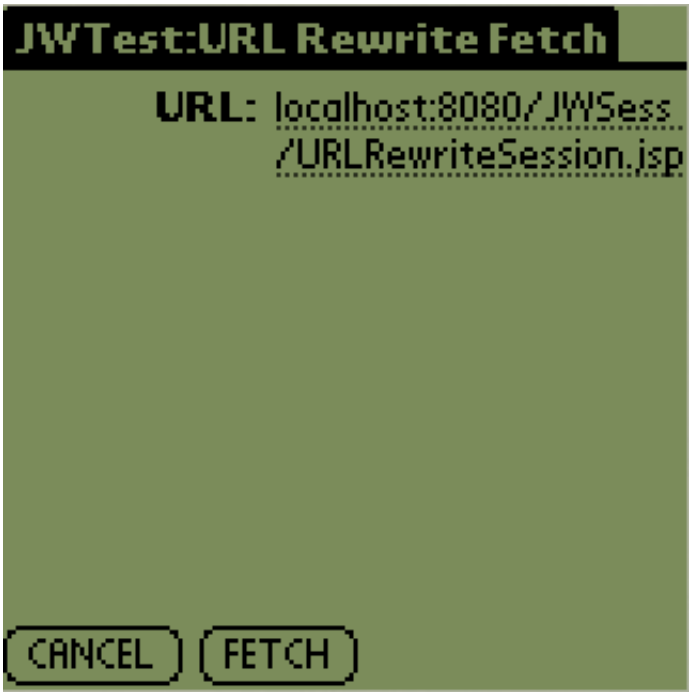

Figure 4. Input a URL for URL rewriting-based session tracking

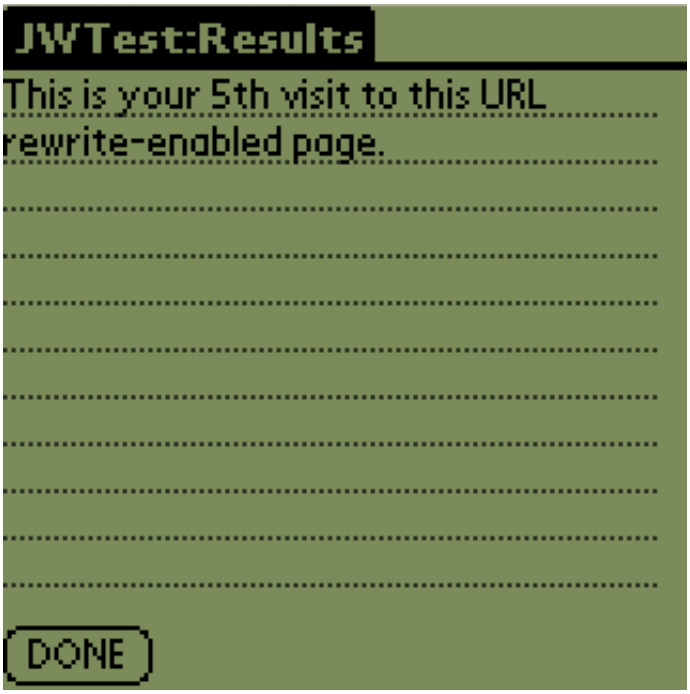

Figure 5. The server page tracks the visit number

Embed complex client state information in XML directly

Session tracking using cookies and URL-rewriting techniques embed session information inside HTTP headers or URLs. However, since the headers or URLs have limited space for embedding complex information, that session-tracking information is usually small pieces of identification strings. In those solutions, the server maintains the real session state information in data objects associated with those identification strings.

However, what if we want richer state information on the client side? We could embed rich session information directly in the *content* exchanged by HTTP connections.

The world of HTML browsers doesn't widely employ this technique—except, perhaps, in hidden fields embedded in HTML forms. Traditional browsers usually require a form to submit content data to the server, and the HTML language itself is designed to describe visual display elements for human beings.

But in the world of MIDP wireless applications, MIDlets handle the visual display UI (user interface) elements. You can optimize the communication between MIDlets and the backend servers to best express application logic using a most efficient XML format. Clearly part of the application logic, session information should be incorporated into the XML communication protocol.

Session information envelope for XML documents

You can add session information to XML documents in two ways: You can place the XML elements containing session information anywhere inside the original XML document (embedded session elements). Or you can place the original document inside a wrapper document, which also contains the session information elements (enveloping session elements).

To keep the original document intact and the parsing simple, we decided to use enveloping session elements. As mentioned above, the main advantage of expressing session information in XML elements

is that the client can maintain rich state information. However, to keep things simple, in our example, we exchange only one SessionID element between the client and server. Our XML document takes the following format:

```
<SessionWrapper>
  <SessionID id="the session id" />
  <OriginalContent>
    <!-- Original content go here -->
  </OriginalContent>
</SessionWrapper>
```

Server side

On the server side, the persistent data objects containing session state information are stored with SessionIDs as attribute pairs in the PageContext object. Those pairs have the *application* scope, which means they can be retrieved as long as the server is running—unless a servlet invalidates them.

If the server receives an XML document with an empty SessionID, it generates a new one according to the current time in milliseconds and embeds it in the return XML document.

Below is the code snippet from our XMLSession.jsp; it illustrates the server-side actions above:

```
// Counter object to be stored with the session.
String count;

// ... ...
// Get session ID sid from the request. That requires some XML parsing
// which will be discussed in the next section.
// ... ...

// Get the "count" attribute associated with the sessionID.
Object o = pageContext.getAttribute(sid, PageContext.APPLICATION_SCOPE);
// If the sessionID is not found,
// set count to zero and start a new session.
if ( o == null ) {
  sid = Long.toString( (new Date()).getTime() );
  count = "0";
  pageContext.setAttribute(sid, count, PageContext.APPLICATION_SCOPE);
} else {
  count = (String) o;
}
// Increase count by one in every visit.
count = Integer.toString(Integer.parseInt(count)+1);
pageContext.setAttribute(sid, count, PageContext.APPLICATION_SCOPE);
```

Client side

In this section, we discuss how to handle the extra XML session elements on the client side. The MIDP client relies on help class XMLSessionWrapper to preprocess XML contents to be posted to the server and post-process XML contents received from the server. The session identification information is stored as a static data member in the XMLSessionWrapper class.

XMLSessionWrapper.unwrapDocument() uses the kXML lightweight CLDC (Connected Limited Device Configuration)-compatible parser to parse the incoming document into a kDOM (Document Object Model) object and then retrieves the session identification string (SessionID). It returns the

unenveloped original XML node for future processing. The following code segment illustrates this process:

```
// Unwrap an input stream "xmlStream",
// which contains a sessionID wrapper around the original data.
// Return a Node representing the unwrapped data.
public static Node unwrapDocument(InputStream xmlStream)
                                  throws Exception {
  InputStreamReader reader = new InputStreamReader(xmlStream);
  XmlParser parser = new XmlParser (reader);
  Document doc = new Document();
  doc.parse(parser);

  // Root element of the wrapped XML document.
  Element wrapper = doc.getRootElement();
  // The first element under root contains session info.
  Element sessionInfo = (Element) wrapper.getChild(0);
  // The second element under root is the original XML document.
  Element originalContent = (Element) wrapper.getChild(1);

  // Set static variable sessionID.
  sessionID = sessionInfo.getAttribute("id").getValue();

  // The unwrapped original content is a node under OriginalContent.
  return (Node) originalContent.getChild(0);
}
```

The XMLSessionWrapper.wrapDocument() method envelops any XML string with SessionID information directly through string manipulations. The following code segment shows the source for the wrapDocument() method:

```
// Embed sessionID into a wrapper XML document around xmlDoc
// and return the wrapper XML document.
public static String wrapDocument(String xmlDoc) {
  String wrapID;
  if ( sessionID == null ) {
    wrapID = "";
  } else {
    wrapID = sessionID;
  }
  return "<SessionWrapper><SessionID id=\"" +
        wrapID +
        "\"/>" + xmlDoc + "<OriginalContent>" +
        "</OriginalContent></SessionWrapper>";
}
```

Figures 6 and 7 show the XML-based session tracking with MIDlet XMLFetch.java and JSP XMLSession.jsp in action.

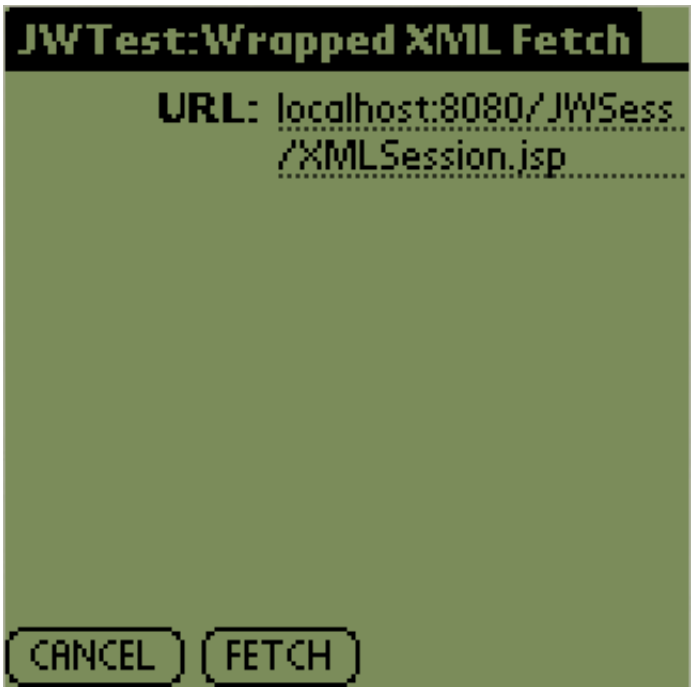

Figure 6. Input a URL for XML-based session tracking. This technique requires close cooperation between the server and the client.

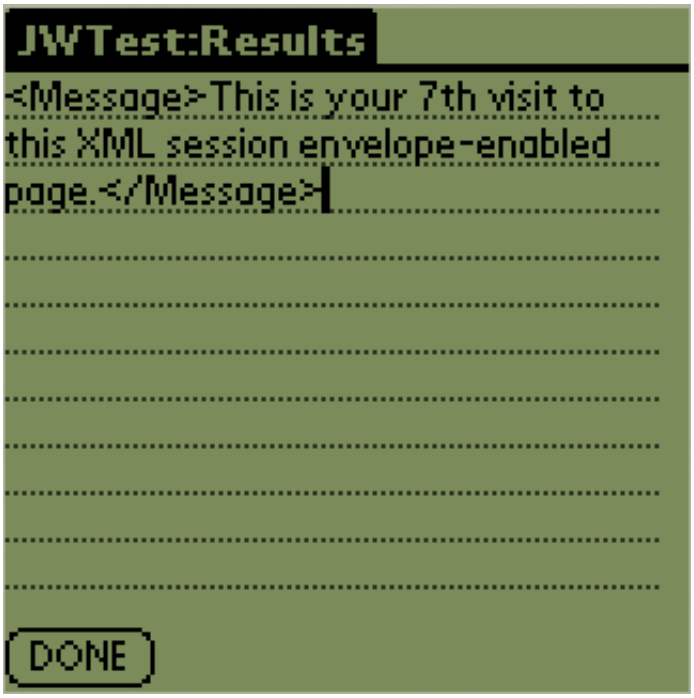

Figure 7. The server page tracks the visit number and returns an XML element containing that information

The right track

In this article, we reviewed three HTTP session-tracking methods for MIDP applications and provided implementation frameworks. The techniques allow us to maintain client state information over the stateless HTTP protocol and therefore prove crucial to enterprise-level MIDP applications.

Our three approaches have different levels of flexibility, power, and transparency. You could use our sample code for the cookie and URL-rewriting techniques directly to add simple out-of-the-box HTTP session support for MIDP applications. We intended our sample code for XML-enveloping session support to show how to exchange rich session state information maintained on the client side. You could adapt that method to tailor each project's individual needs.

About the author

Michael Yuan and Ju Long are PhD candidates at the University of Texas at Austin. They use J2ME and J2EE to develop mobile research applications for projects in the Center for Research in Electronic Commerce.

Resources

- Download the source code of this article's example application as well as the accompanying MIDlets and JSPs: http://www.javaworld.com/javaworld/jw-04-2002/wireless/jw-0426-wireless.zip
- For more information on running this article's example programs, see "Deploy the Example MIDP Application on Palm OS Devices," a sidebar from the authors' previous *JavaWorld* article "Build Database-Powered Mobile Applications on the Java Platform" (January 2002): http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-midp.html#sidebar1
- "Build Database-Powered Mobile Applications on the Java Platform" (*JavaWorld*, January 2002)

introduces the architecture and designs of MIDP client/server applications: http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-midp.html
- For a MIDP programming refresher, read Michael Cymerman's series "Device Programming with MIDP" (*JavaWorld*):
  - Part 1: The concepts behind MIDP APIs and J2ME to build cross-*wireless*-platform apps (January 2001)
  - Part 2: Use these user-interface and data-store components to create MIDP-based applications (March 2001)
  - Part 3: Use MIDP's communication APIs to interact with external systems (July 2001)
- The "Sun Smart Ticket Demo" is Sun's showcase for J2ME/J2EE applications: http://java.sun.com/blueprints/code/jsm10/application.html
- Tutorials on session tracking using HttpSession:
  - From *The Java Tutorial*, Mary Campione, Kathy Walrath, Alison Huml (Addison-Wesley, 2000; ISBN: 0201703939): http://java.sun.com/docs/books/tutorial/servlets/client-state/session-tracking.html
  - From *Core Servlets and JavaServer Pages*, Marty Hall (Sun Microsystems Press and Prentice Hall, May 2000; ISBN: 0130893404) http://www.apl.jhu.edu/~hall/java/Servlet-Tutorial/Servlet-Tutorial-Session-Tracking.html
- RFC 2109 "HTTP State Management Mechanism" defines how to use cookies to track HTTP sessions (Netscape Communications, 1997): http://www.ietf.org/rfc/rfc2109.txt
- RFC 2695 (the Internet Society, 2000) updates RFC 2109: http://www.ietf.org/rfc/rfc2965.txt
- For a lightweight general-purpose library devoted to cookie handling on the device, read "A Recipe for Cookie Management," Sonal Bansal (*JavaWorld*, April 2002): http://www.javaworld.com/javaworld/jw-04-2002/jw-0426-cookie.html
- The kXML parser project: http://kxml.enhydra.org/
- Browse the **Wireless Development** section of *JavaWorld*'s Topical Index: http://www.javaworld.com/channel_content/jw-wireless-index.shtml
- Browse the **Java 2 Platform, Micro Edition (J2ME)** section of *JavaWorld*'s Topical Index: http://www.javaworld.com/channel_content/jw-j2me-index.shtml
- Chat about devices galore in *JavaWorld*'s **Device Programming** discussion: http://forums.idg.net/webx?50@@.ee6b808
- Sign up for *JavaWorld*'s free weekly email newsletters: http://www.javaworld.com/jw-subscribe
- You'll find a wealth of IT-related articles from our sister publications at IDG.net