Maintaining state in JAX-WS Web Services

Kai Qian

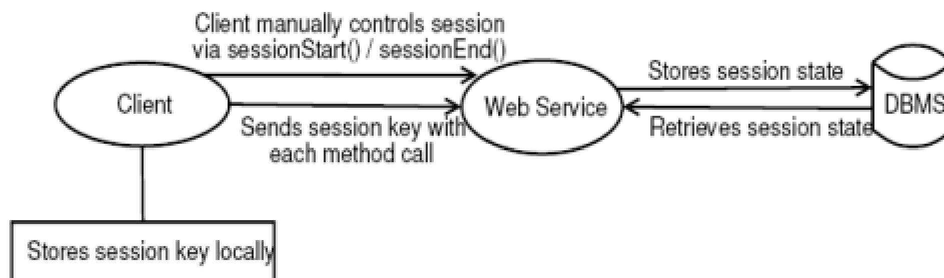
Internal Report
July, 2008

School of Computing and Software Engineering
Southern Polytechnic State University
Marietta, Georgia, USA

The purpose of this paper is to examine various options for maintaining state in web services. It will look at JAX-WS web services in particular to demonstrate the methods in order to compare and contrast them according to their strengths and weaknesses.

Method 1: Manually maintaining state

A web service implementing this method of state management would have to take on the responsibility for tracking and maintaining the conversational state. In addition, the client would also have to be aware of the mechanism (or at least that the mechanism exists). The session id would become essentially another parameter to each method that participates in or access the session. The web service could then save the state on the server by either writing it to a file or a database via JDBC



or the Java Persistence API.

Sample web service code:

```
@WebService
public class SWS {
    public int sessionStart() {
        int key = (int) (Math.random() * 1000);
        Connection con = null;
        try {
            Class.forName("org.postgresql.Driver");
            con = DriverManager.getConnection("jdbc:postgresql:wstest", "user", "pass");
            Statement stmt = con.createStatement();
            String query = "insert into wsstate (key) values (" + key + ")";
            stmt.executeUpdate(query);
```

```
        }
        catch(Exception e) {
            e.printStackTrace();
        }
        finally {
            try { if(con != null) { con.close(); } } catch(Exception e) { }
            return key;
        }
    }

    public void sessionEnd(int key) {
        Connection con = null;
        try {
            Class.forName("org.postgresql.Driver");
            con = DriverManager.getConnection("jdbc:postgresql:wstest", "user", "pass");
            Statement stmt = con.createStatement();
            String query = "delete from wsstate where key = " + key;
            stmt.executeUpdate(query);
        }
        catch(Exception e) {
            e.printStackTrace();
        }
        finally {
            try { if(con != null) { con.close(); } } catch(Exception e) { }
        }
    }

    public double getAverage(int key, int n) {
        Connection con = null;
        double average = 0.0;
        try {
            Class.forName("org.postgresql.Driver");
            con = DriverManager.getConnection("jdbc:postgresql:wstest", "user", "pass");
            Statement stmt = con.createStatement();
            String query = "select * from wsstate where key = " + key;
            ResultSet rslt = stmt.executeQuery(query);
            rslt.next();
            int count = rslt.getInt("count");
            int amount = rslt.getInt("amount");
            amount += n;
            count ++;
            average = ((double) amount) / count;
            query = "update wsstate set count = " + count + ", amount = " + amount + " where 
key = " + key;
            stmt.executeUpdate(query);
        }
        catch(Exception e) {
            e.printStackTrace();
        }
        finally {
            try { if(con != null) { con.close(); } } catch(Exception e) { }
            return average;
        }
    }
}
```

Sample client:

```
public class Main {
    public static void main(String[] args) {
        try {
            com.datavikings.sws.SWSService service = new com.datavikings.sws.SWSService();
            com.datavikings.sws.SWS port = service.getSWSPort();
            int key = port.sessionStart();
            Scanner scan = new Scanner(System.in);
            System.out.print("> ");
            int input = scan.nextInt();
```

```
            while(input != 0) {
                double avg = port.getAverage(key, input);
                System.out.println("Average: " + avg);
                System.out.print("> ");
                input = scan.nextInt();
            }
            port.sessionEnd(key);
        }
        catch (Exception ex) { }
    }
}
```

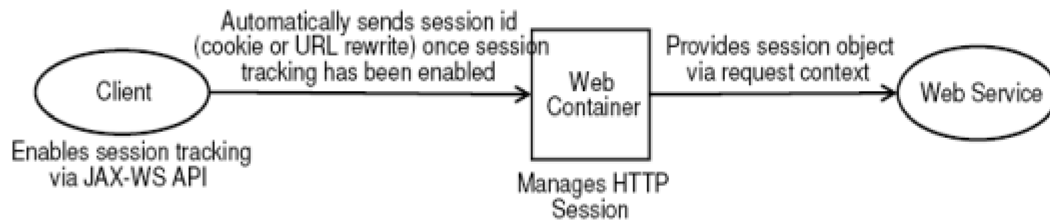
This method has some obvious drawbacks. The client must be aware of the session tracking mechanism and must actively participate. Also, a lot of work needs to be done on the server in order to save and retrieve the session state. This example was fairly straightforward, but the amount of work involved could become a headache for even moderately complex applications. The session id is chosen randomly here and is prone to collisions (two people being assigned the same session id accidentally). While steps can be taken to make id collisions less likely, even a remote chance is unacceptable in a production environment, so other mechanisms would have to be put in place to ensure that id's are unique. The client has ready access to the session id and could very easily change it accidentally, causing the session mechanism to break. A client's IP address might be chosen instead of a random value for the session ID, which could be checked against incoming requests to add a cursory layer of security. This approach would be inadequate for modern networks, however, many of which rely on Network Address Translation (NAT) to effectively hide multiple hosts behind a single IP address.

Method 2: Accessing the underlying HTTP session via MessageContext

The javax.xml.ws.handler.MessageContext interface allows for retrieval of information about the request that caused the web method invocation. The MessageContext is obtained by calling the getMessageContext() method of javax.xml.ws.WebServiceContext. Marking a member variable of type WebServiceContext in the web service class with the javax.annotation.Resource (@Resource) annotation causes a WebServiceContext object to be injected into the class. Once the MessageContext has been obtained, the HttpServletRequest object that represents the client HTTP request can be obtained by calling the MessageContext's get(MessageContext.SERVLET_REQUEST) method. Once the request object is obtained, the HTTP session can be obtained with the request object's .getSession() method. Any state that needs to be maintained can be stored in the session object by means of the setAttribute() method and retrieved later with the getAttribute() method, similar to the way servlets maintain session-specific data.

To enable sessions in the client, the SESSION_MAINTAIN_PROPERTY property of the port's request context must be set to true. This property tells the client to maintain the session with the web service by keeping the session information returned by the web service. By default the session information is ignored. By accessing the .getRequestContext() method of the BindingProvider interface (an explicit cast is needed), a Map containing the request context properties can be obtained and the property set to true.

The session information may take the form of cookies or rewriting part of the URL to include the session id. One of the advantages to allowing the web container to control the session handling is that the code does not have to be modified to switch from cookies to URL rewriting for session handling. The web container can be configured to use the preferred method without recompiling any code.



Sample web service code:

```
@WebService
public class SWS {
    @Resource
    private WebServiceContext ctx;
    public double getAverage(int n){
        HttpSession session = getSession();
        if (session == null) {
            throw new WebServiceException("Something went wrong!");
             }
        Integer count = (Integer) session.getAttribute("count");
             Integer amount = (Integer) session.getAttribute("amount");
        if (count == null || amount == null) {
            count = new Integer(0);
                    amount = new Integer(0);
            System.out.println("Starting the Session");
        }
        count++;
        amount += n;
        session.setAttribute("count", count);
             session.setAttribute("amount", amount);
        return ((double) amount) / count;
    }

    private HttpSession getSession() {
        MessageContext mc = ctx.getMessageContext();
        HttpServletRequest request = (javax.servlet.http.HttpServletRequest)
mc.get(MessageContext.SERVLET_REQUEST);
        HttpSession session = request.getSession();
        return session;
    }
}
```

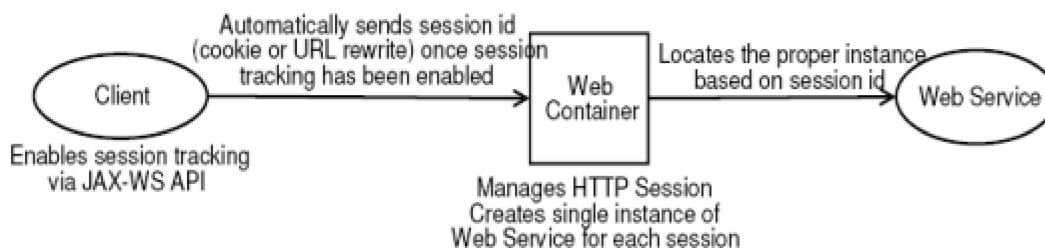
Sample client code:

```
public class Main {
  public static void main(String[] args) {
    try {
      com.datavikings.sws.SWSService service = new com.datavikings.sws.SWSService();
      com.datavikings.sws.SWS port = service.getSWSPort();
      ((BindingProvider)
port).getRequestContext().put(BindingProvider.SESSION_MAINTAIN_PROPERTY, true);
       Scanner scan = new Scanner(System.in);
       System.out.print("> ");
       int input = scan.nextInt();
       while(input != 0) {
         double avg = port.getAverage(input);
         System.out.println("Average: " + avg);
         System.out.print("> ");
```

```
          input = scan.nextInt();
        }
      }
      catch (Exception e) {
        e.printStackTrace();
      }
    }
}
```

Method 3: Using the @HttpSessionScope annotation

This method requires less coding and relies on annotations to do most of the heavy lifting. It is similar to method 2 in that it relies on the web container to handle the session management, but the programmer does not have to explicitly perform any storing of the state in an HttpSession object. The @HttpSessionScope annotation tells the web container to keep a unique instance of the web service for each client. The following sample code shows the use of the @HttpSessionScope annotation to build a web service.



Sample code:

```
@HttpSessionScope @Stateful @WebService
public class SWS {
	private int value;

	public void addAmount(int amount) {
		value += amount;
	}

	public int getValue() {
		return value;
	}
}
```

The example clearly demonstrates the advantage of the @HttpSessionScope annotation. The code is much simpler, meaning it is easier to write, read, and maintain. It retains all of the advantages of method 2 without any of the drawbacks. The addition of a single annotation grants all of the functionality without having to explicitly retrieve or store objects in a session context.

Without the @HttpSessionScope annotation, the web service will maintain the int value members across invocations from any client. The annotation binds the instance variables to the scope of the client's session. It is important to note that the @Stateful annotation is from the com.sun.xml.ws.developer package, not to be confused with the @Stateful annotation in the javax.ejb package.

Conclusion

Method 1 produces the desired result but introduces unwanted complexity. The client must also be an active participant in the session tracking in order for this method to work. The web service must also manage the session id's to ensure that collisions to not occur.

Method 2 does a much better job of maintaining state because it leaves the details of session creation and maintenance to the web container. This method does require a good bit of setup in order to get started.

Of the three methods examined, method 3 is by far the easiest to use. With a single annotation it provides the desired functionality without introducing unwanted side-effects. Like method 2, the session details are left up to the web container. With the exception of enabling session tracking property in the request context, the client remains largely ignorant of any session tracking mechanism details.