Java theory and practice: Are all stateful Web applications broken?

HttpSession and friends are trickier than they look

Brian Goetz

September 23, 2008

The session state management mechanism provided by the Servlets framework, HttpSession, makes it easy to create stateful applications, but it is also quite easy to misuse. Many Web applications that use HttpSession for mutable data (such as JavaBeans classes) do so with insufficient coordination, exposing themselves to a host of potential concurrency hazards.

View more content in this series

While there are many Web frameworks in the Java™ ecosystem, they all are based, directly or indirectly, on the Servlets infrastructure. The Servlets API provides a host of useful features, including state management through the HttpSession and ServletContext mechanisms, which allows the application to maintain state that persists across multiple user requests. However, some subtle (and largely unwritten) rules govern the use of shared state in Web applications, of which many applications unknowingly fall afoul. The result is that many stateful Web applications have subtle and serious flaws.

Scoped containers

The ServletContext, HttpSession, and HttpRequest objects in the Servlet specification are referred to as scoped containers. Each of these has getAttribute() and setAttribute() methods, which store data on behalf of the application. The difference between them is the lifetime of the scoped container. For HttpRequest, the data only persists for the lifetime of the request; for HttpSession, it persists for the lifetime of a session between a user and the application; and for ServletContext, it persists for the lifetime of the application.

Because the HTTP protocol is stateless, scoped containers are tremendously useful in the construction of stateful Web applications; the servlet container takes responsibility for managing application state and data life cycle. While the specification is largely silent on the subject, the session- and application-scoped containers must also to some degree be thread-safe, because the getAttribute() and setAttribute() methods may be called at any time by different threads. (The
specification does not directly mandate that these implementations be thread-safe, but the nature of the service they provide effectively requires it.)

Scoped containers also offer another potentially significant benefit to Web applications: the container can manage replication and fail-over of application state transparently to the application.

Sessions

A session is a series of request-response exchanges between a specific user and a Web application. Users expect that Web sites will remember their authentication credentials, the contents of their shopping cart, and information entered in Web forms on previous requests, but the core HTTP protocol is stateless, meaning that all the information about a request must be stored in the request itself. So to create useful interactions with users with a duration of longer than a single request-response cycle, session state must be maintained somewhere. The servlet framework allows each request to be associated with a session and provides the HttpSession interface to act as a value store for (key, value) data items relevant to that session. Listing 1 shows a typical bit of servlet code that stores shopping cart data in the HttpSession:

Listing 1. Using HttpSession to store shopping cart information

```java
HttpSession session = request.getSession(true);
ShoppingCart cart = (ShoppingCart)session.getAttribute("shoppingCart");
if (cart == null) {
    cart = new ShoppingCart(...);
    session.setAttribute("shoppingCart");
}
doSomethingWith(cart);
```

The usage in Listing 1 is typical for servlets; the application looks to see if an object has already been placed in the session, and if not, it creates one that can be used by subsequent requests on that session. Web frameworks built atop servlets (such as JSP, JSF, SpringMVC, and so on) hide the details but essentially perform this same sort of operation on your behalf for data that is tagged as session-scoped. Unfortunately, the usage in Listing 1 is also likely to be incorrect.

Threading considerations

When an HTTP request arrives at the servlet container, HttpServletRequest and HttpServletResponse objects are created and passed to the service() method of a servlet, in the context of a thread managed by the servlet container. The servlet is responsible for producing the response; the servlet maintains control of that thread until the response is complete, at which point the thread is returned to the pool of available worker threads. Servlet containers maintain no affinity between threads and sessions; the next request to come in on a given session will likely be serviced by a different thread than the current request. In fact, it is possible for multiple simultaneous requests to come in on the same session (which can happen in Web applications that use frames or AJAX techniques to fetch data from the server while the user is interacting with the page). In this case, there can be multiple simultaneous requests from the same user executing concurrently on different threads.

Most of the time, threading considerations like these are irrelevant to the Web application developer. The stateless nature of HTTP encourages that the response be a function only of data stored in the request (which is not shared with other concurrent requests) and data stored in
repositories (such as databases) that already manage concurrency control. However, once a Web application stores data in a shared container like HttpSession or ServletContext, we've turned our Web application into a concurrent one, and we now have to think about thread-safety within the application.

While thread-safety is a term we typically use to describe code, in actuality it is about data. Specifically, thread safety is about properly coordinating access to mutable data that is accessed by multiple threads. Servlet applications are frequently thread-safe by virtue of the fact that they do not share any mutable data and therefore require no additional synchronization. But there are lots of ways that shared state can be introduced into Web applications — not only scoped containers like HttpSession and ServletContext, but also static fields and instance fields of HttpServletRequest objects. Once a Web application wants to share data across requests, the application developer must pay attention to where that shared data is and ensure that there is sufficient coordination (synchronization) between threads when accessing the shared data to avoid threading hazards.

Threading risks for Web applications

When a Web application stores mutable session data such as a shopping cart in an HttpSession, it becomes possible that two requests may try to access the shopping cart at the same time. Several failure modes are possible, including:

- An atomicity failure, where one thread is updating multiple data items and another thread reads the data while they are in an inconsistent state
- A visibility failure between a reading thread and a writing thread, where one thread modifies the cart but the other sees a stale or inconsistent state for the cart's contents

Atomicity failures

Listing 2 shows a (broken) implementation of methods for setting and retrieving the high scores in a gaming application. It uses a PlayerScore object to represent the high score, which is an ordinary JavaBean class with the properties name and score, stored in the application-scoped ServletContext. (It is assumed that, at application startup, the initial high score is installed as the highScore attribute in the ServletContext, so the getAttribute() calls will not fail.)

Listing 2. Broken scheme for storing related items in a scoped container

```java
public PlayerScore getHighScore() {
    ServletContext ctx = getServletConfig().getServletContext();
    PlayerScore hs = (PlayerScore) ctx.getAttribute("highScore");
    PlayerScore result = new PlayerScore();
    result.setName(hs.getName());
    result.setScore(hs.getScore());
    return result;
}

public void updateHighScore(PlayerScore newScore) {
    ServletContext ctx = getServletConfig().getServletContext();
    PlayerScore hs = (PlayerScore) ctx.getAttribute("highScore");
    if (newScore.getScore() > hs.getScore()) {
        hs.setName(newScore.getName());
        hs.setScore(newScore.getScore());
    }
}
```

Java theory and practice: Are all stateful Web applications broken?
A number of things about the code in Listing 2 are broken. The approach taken here is to store a mutable holder for the high scoring player's name and score in the \texttt{ServletContext}. When a new high score is reached, both the name and score must be updated.

Suppose the current high scoring player is Bob, with a score of 1000, and his score is beaten by Joe, with a score of 1100. Near the time at which Joe's score is being installed, another player requests the high score. The \texttt{getHighScore()} method will retrieve the \texttt{PlayerScore} object from the servlet context and fetch the name and score from it. With some unlucky timing, though, it is possible to retrieve Bob's name and Joe's score, showing Bob to have achieved a score of 1100, something that never happened. (This failure might be acceptable for a free game site, but replace "score" with "bank balance" and it seems less harmless.) This is an atomicity failure, in that two operations that are supposed to be atomic with respect to each other — fetching the name/score pair and updating the name/score pair — did not in fact execute atomically with respect to each other, and one of the threads was allowed to see the shared data in an inconsistent state.

Further, because the score-updating logic follows the \texttt{check-then-act} pattern, it is possible for two threads to "race" to update the high score, with unpredictable results. Suppose the current high score is 1000, and two players simultaneously register high scores of 1100 and 1200. With some unlucky timing, both will pass the test of "is new score higher than existing high score," and both will enter the block that updates the high score. Again, depending on timing, the outcome might be inconsistent (the name of one player and the high score of the other), or just wrong (the player scoring 1100 could overwrite the name and score of the player scoring 1200).

\section*{Visibility failures}

More subtle than atomicity failures are \textit{visibility failures}. In the absence of synchronization, if one thread writes to a variable and another thread reads that same variable, the reading thread could see \textit{stale}, or out-of-date, data. Worse, it is possible for the reading thread to see up-to-date data for variable \texttt{x} and stale data for variable \texttt{y}, even if \texttt{y} was written before \texttt{x}. Visibility failures are subtle because they don't happen predictably, or even frequently, causing rare and difficult-to-debug intermittent failures. Visibility failures are created by data races — failure to properly synchronize when accessing shared variables. Programs with data races are, for all intents and purposes, broken, in that their behavior cannot be reliably predicted.

The Java Memory Model (JMM) defines the conditions under which a thread reading a variable is guaranteed to see the results of a write in another thread. (A full explanation of the JMM is beyond the scope of this article; see \textit{Related topics}.) The JMM defines an ordering on the operations of a program called \textit{happens-before}. Happens-before orderings across threads are only created by synchronizing on a common lock or accessing a common volatile variable. In the absence of a happens-before ordering, the Java platform has great latitude to delay or change the order in which writes in one thread become visible to reads of that same variable in another.

The code in \texttt{Listing 2} has visibility failures as well as atomicity failures. The \texttt{updateHighScore()} method retrieves the \texttt{HighScore} object from the \texttt{ServletContext} and then modifies the state of the \texttt{HighScore} object. The intent is for those modifications to be visible to other threads that call \texttt{getHighScore()}, but in the absence of a happens-before ordering between the writes to the name
and score properties in updateHighScore() and the reads of those properties in other threads calling getHighScore(), we are relying on good luck for the reading threads to see the correct values.

Possible solutions

While the servlet specification does not adequately describe the happens-before guarantees that a servlet container must provide, one is forced to conclude that placing an attribute in a shared scoped container (HttpSession or ServletContext) happens before another thread retrieves that same attribute. (See JCiP 4.5.1 for the reasoning behind this conclusion. All the specification says is "Multiple servlets executing request threads may have active access to a single session object at the same time. The Developer has the responsibility for synchronizing access to session resources as appropriate.")

The set-after-write trick

It is a commonly cited "best practice" that when updating mutable data stored in scoped session containers, one must call setAttribute() again after modifying the data. Listing 3 shows an example of updateHighScore() rewritten to use this technique. (One of the motivations for this technique is to hint to the container that the value has been changed, so that the session or application state can be resynchronized across instances in a distributed Web application.)

Listing 3. Using the set-after-write technique to hint to the servlet container that the value has been updated

```java
public void updateHighScore(PlayerScore newScore) {
    ServletContext ctx = getServletConfig().getServletContext();
    PlayerScore hs = (PlayerScore) ctx.getAttribute("highScore");
    if (newScore.getScore() > hs.getScore()) {
        hs.setName(newScore.getName());
        hs.setScore(newScore.getScore());
        ctx.setAttribute("highScore", hs);
    }
}
```

Unfortunately, while this technique helps with the problem of efficiently replicating session and application state in clustered applications, it is not enough to fix the basic thread-safety problems in our example. It is enough to mitigate the visibility problems (that another player might never see the values updated in updateHighScore()), but it is not enough to address the multiple potential atomicity problems.

Piggybacking on synchronization

The set-after-write technique is able to eliminate the visibility problems because the happens-before ordering is transitive, and there is a happens-before edge between the call to setAttribute() in updateHighScore() and the call to getAttribute() in getHighScore(). Because the updates to the HighScore state happen before setAttribute(), which happens before the return from getAttribute(), which happens before the use of the state by the caller of getHighScore(), transitivity lets us conclude that the values seen by callers of getHighScore()
are at least as up to date as the most recent call to `setAttribute()`. This technique is called *piggybacking on synchronization*, because the `getHighScore()` and `updateHighScore()` methods are able to use their knowledge of synchronization in `getAttribute()` and `setAttribute()` to provide some minimal guarantees of visibility. However, in the example as written, it is still not enough. The set-after-write technique may be useful for state replication, but it is not enough to provide thread safety.

**Leaning on immutability**

A useful technique for creating thread-safe applications is to lean on immutable data as much as possible. Listing 4 shows our high score example rewritten to use an *immutable* implementation of `HighScore` that is free of the atomicity failures that would allow a caller to see a nonexistent player/score pair, as well as the visibility failures that would prevent a caller of `getHighScore()` from seeing the most recent values written by a call to `updateHighScore()`:

**Listing 4. Using an immutable HighScore object to close most of the atomicity and visibility holes**

```java
public class HighScore {
    public final String name;
    public final int score;

    public HighScore(String name, int score) {
        this.name = name;
        this.score = score;
    }
}

public PlayerScore getHighScore() {
    ServletContext ctx = getServletConfig().getServletContext();
    return (PlayerScore) ctx.getAttribute("highScore");
}

public void updateHighScore(PlayerScore newScore) {
    ServletContext ctx = getServletConfig().getServletContext();
    PlayerScore hs = (PlayerScore) ctx.getAttribute("highScore");
    if (newScore.score > hs.score)
        ctx.setAttribute("highScore", newScore);
}
```

The code in Listing 4 has many fewer potential failure modes. Piggybacking on the synchronization in `setAttribute()` and `getAttribute()` guarantees visibility. The fact that only a single immutable data item is being stored eliminates the potential atomicity failure that a caller to `getHighScore()` could see an inconsistent update to the name/score pair.

Placing immutable objects in a scoped container avoids most atomicity and visibility failures; it is also safe to place *effectively immutable* objects in a scoped container. Effectively immutable objects are those that, while theoretically mutable, are never actually modified after being published, such as a JavaBean whose setters are never called after placing the object in an `HttpSession`.

Data placed in an `HttpSession` is not only accessed by the requests on that session; it may also be accessed by the container itself if the container is doing any sort of state replication.
All data placed in an HttpSession or ServletContext should be thread-safe or effectively immutable.

**Effecting atomic state transitions**

The code in **Listing 4** still has one problem, though — the check-then-act in `updateHighScore()` still enables a potential race between two threads trying to update the high score. With some unlucky timing, an update could be lost. Two threads could pass the "is the new high score greater than the old one" check at the same time, causing both to call `setAttribute()`. Depending on timing, there is no guarantee that the higher of these two scores will win. To close this last hole, we need a means of atomically updating the score reference while guaranteeing freedom from interference. Several approaches can be used to do so.

Listing 5 adds synchronization to `updateHighScore()` to ensure that the check-then-act inherent in the update process cannot execute concurrently with another update. This approach is adequate provided that all such conditional modification logic acquire the same lock used by `updateHighScore()`.

**Listing 5. Using synchronization to close the last atomicity hole**

```java
public void updateHighScore(PlayerScore newScore) {
    ServletContext ctx = getServletConfig().getServletContext();
    synchronized (lock) {
        PlayerScore hs = (PlayerScore) ctx.getAttribute("highScore");
        if (newScore.score > hs.score)
            ctx.setAttribute("highScore", newScore);
    }
}
```

While the technique in Listing 5 works, there is an even better technique: use the `AtomicReference` class in the `java.util.concurrent` package. This class is designed to provide atomic conditional updates through the `compareAndSet()` call. Listing 6 shows how to use an `AtomicReference` to restore this last bit of atomicity to our example. This approach is preferable to the code in Listing 5 because it is harder to accidentally violate the assumptions about how to update the high score.

**Listing 6. Using an AtomicReference to close the last atomicity hole**

```java
public PlayerScore getHighScore() {
    ServletContext ctx = getServletConfig().getServletContext();
    AtomicReference<PlayerScore> holder
        = (AtomicReference<PlayerScore>) ctx.getAttribute("highScore");
    return holder.get();
}

public void updateHighScore(PlayerScore newScore) {
    ServletContext ctx = getServletConfig().getServletContext();
    AtomicReference<PlayerScore> holder
        = (AtomicReference<PlayerScore>) ctx.getAttribute("highScore");
    while (true) {
        HighScore old = holder.get();
        if (old.score >= newScore.score)
            break;
        else if (holder.compareAndSet(old, newScore))
            break;
    }
}
```
For mutable objects placed in scoped containers, their state transitions should be made atomic, either through synchronization or through the atomic variable classes in `java.util.concurrent`.

**Serializing access to an HttpSession**

In the examples I've given so far, I've tried to avoid the various hazards associated with accessing data in the application-wide `ServletContext`. It is clear that careful coordination is required when accessing the `ServletContext`, because the `ServletContext` is accessible from any request. Most stateful Web applications, however, lean more heavily on the session-scoped container, `HttpSession`. It may not be obvious how multiple simultaneous requests could happen on the same session; after all, a session is tied to a particular user and browser session, and users might not seem to request multiple pages at once. But requests on a session can overlap in applications that generate requests programmatically, such as AJAX applications.

Requests on a single session can indeed overlap, and this ability is unfortunate. If requests on a session could be easily serialized, nearly all the hazards described here would not be an issue when accessing shared objects in an `HttpSession`; serialization would prevent the atomicity failures, and piggybacking on the synchronization implicit in `HttpSession` would prevent the visibility failures. And serializing requests tied to a specific session is unlikely to impose any significant impact on throughput, as it is somewhat rare to have requests on a session overlap at all, and it is quite rare to have many requests on a session overlap.

Unfortunately, there's no option in the servlet specification to say "force requests on the same session to be serialized." However, the SpringMVC framework offers a way to ask for this, and the approach can be reimplemented in other frameworks easily. The base class for SpringMVC controllers, `AbstractController`, provides a boolean variable `synchronizeOnSession`; when this is set, it will use a lock to ensure that only one request on a session executes concurrently.

Serializing requests on an `HttpSession` makes many concurrency hazards go away, in a similar way that confining objects to the Event Dispatch Thread (EDT) reduces the requirement for synchronization in Swing applications.

**Summary**

Many stateful Web applications have significant concurrency vulnerabilities that stem from accessing mutable data stored scoped containers like `HttpSession` and `ServletContext` without adequate coordination. It is easy to mistakenly assume that the synchronization inherent in the `getAttribute()` and `setAttribute()` methods is sufficient — but it only holds true under certain circumstances, such as when the attribute is an immutable, an effectively immutable, or a thread-safe object, or when requests that might access the container are serialized.

In general, everything you place in a scoped container should be effectively immutable or thread-safe. The scoped container mechanism provided by the servlet specification was never intended to manage mutable objects that did not provide their own synchronization. The biggest offender is
storing ordinary JavaBeans classes in an HttpSession. This technique is only guaranteed to work when the JavaBean is never modified after it is stored in the session.
Related topics

- *Java theory and practice* (Brian Goetz, developerWorks): Read the complete series.
- *Java Concurrency in Practice* (Brian Goetz, Addison-Wesley Professional, May 2006): Section 4.5.1 explains the analysis by which one concludes the happens-before guarantees for shared scoped containers. Section 3.5.4 explains the concept of effective immutability. Chapter 16 explains the details of the Java Memory Model and the *happens-before* ordering.
- **Servlet 2.5 specification**: Defines the *HttpSession* and *ServletContext* mechanisms.
- **Spring Framework**: Learn more about SpringMVC.
- **State Replication in the Web Tier**: Explains State replication by servlet containers.

© Copyright IBM Corporation 2008
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)