# MemoryLeakProtection

For some time Tomcat has had some means of protection against memory leaks when stopping or redeploying applications. This page tries to list them, and shows the situations where leaks can be detected and fixed.

## Diagnose a classloader leak upon request

Starting with tomcat 6.0.25, the manager webapp has a new "Find Leaks" button. When triggered, it displays a list of webapps (their context path) that have been stopped (this includes undeployed and redeployed ones) but whose classloader failed to be GCed.

If a leaking webapp is redeployed several times, it will appear as many times as it actually leaked.

**Caution:** This diagnosis calls `System.gc()` which may not be desirable in production environments.

## Different types of leaks that Tomcat can detect (or not)

When a webapp execution is stopped (this encompassed redeploy and undeploy), tomcat tries to detect and fix leaks.

Starting with tomcat 6.0.24, messages are logged to indicate the kind of leak that was detected.

## Summary matrix

| Leak cause | Detected by tomcat | Fixed by tomcat | Possible enhancements |
|---|---|---|---|
| Custom ThreadLocal class | >=6.0.24 | >= 7.0.6 | |
| Webapp class instance as ThreadLocal value | >=6.0.24 | >= 7.0.6 | |
| Webapp class instance indirectly held through a ThreadLocal value | no | >= 7.0.6 | |
| ThreadLocal pseudo-leak | >=6.0.24 | >= 7.0.6 | |
| ContextClassLoader / Threads spawned by webapps | >=6.0.24 | In 6.0.24-6.0.26 `TimerThread` are stopped but it may lead to problems. Optional from 6.0.27 with the `clearReferencesStopTimerThreads` flag. Other threads may be stopped with the `clearReferencesStopThreads` flag, but it's unsafe. | Fix the application to stop the thread when the application is stopped |
| ContextClassLoader / Threads spawned by classes loaded by the common classloader | >=6.0.24 | In 6.0.24-6.0.26 `TimerThread` are stopped but it may lead to problems. Optional from 6.0.27 with the `clearReferencesStopTimerThreads` flag. Other threads may be stopped with the `clearReferencesStopThreads` flag, but it's unsafe. | Fix the offending code (set the correct CCL when spawning the thread) |
| ContextClassLoader / Threads spawned by JRE classes | no | >=6.0.24 pre-spawns some known offenders | |
| static class variables | no | > 6.0.? . Disabled by default with tomcat 7 | |
| LogFactory | | > 6.0.? | |
| JDBC driver registration | > 6.0.? | > 6.0.? | |
| RMI Target | | > 6.0.? | |

### ThreadLocal leaks

Classloader leaks because of uncleaned `ThreadLocal` variables are quite common. Depending on the use cases, they can be detected or not.

#### Custom ThreadLocal class

Suppose we have the following 3 classes in our webapp :

```
public class MyCounter {
        private int count = 0;

        public void increment() {
                count++;
        }

        public int getCount() {
                return count;
        }
}

public class MyThreadLocal extends ThreadLocal<MyCounter> {
}

public class LeakingServlet extends HttpServlet {
        private static MyThreadLocal myThreadLocal = new MyThreadLocal();

        protected void doGet(HttpServletRequest request,
                        HttpServletResponse response) throws ServletException, IOException {

                MyCounter counter = myThreadLocal.get();
                if (counter == null) {
                        counter = new MyCounter();
                        myThreadLocal.set(counter);
                }

                response.getWriter().println(
                                "The current thread served this servlet " + counter.getCount()
                                                + " times");
                counter.increment();
        }
}
```

If the `LeakingServlet` is invoked at least once and the Thread that served it is not stopped, then we created a classloader leak !

The leak is caused because we have a custom class for the `ThreadLocal` instance, and also a custom class for the value bound to the Thread. Actually the important thing is that both classes were loaded by the webapp classloader.

Hopefully tomcat 6.0.24 can detect the leak when the application is stopped: each Thread in the JVM is examined, and the internal structures of the Thread and `ThreadLocal` classes are introspected to see if either the `ThreadLocal` instance or the value bound to it were loaded by the `WebAppClassLoader` of the application being stopped.

In this particular case, the leak is detected and a message is logged. Tomcat 6.0.24 to 6.0.26 modify internal structures of the JDK (`ThreadLocalMap`) to remove the reference to the `ThreadLocal` instance, but this is unsafe (see #48895) so that it became optional and disabled by default from 6.0.27. Starting with Tomcat 7.0.6, the threads of the pool are renewed so that the leak is safely fixed.

```
Mar 16, 2010 11:47:24 PM org.apache.catalina.loader.WebappClassLoader clearThreadLocalMap
SEVERE: A web application created a ThreadLocal with key of type [test.MyThreadLocal] (value [test.
MyThreadLocal@4dbb9a58]) and a value of type [test.MyCounter] (value [test.MyCounter@57922f46]) but failed to
remove it when the web application was stopped. To prevent a memory leak, the ThreadLocal has been forcibly
removed.
```

**Note**: this particular leak was actually already cured by previous versions of tomcat 6, because static references of classes loaded by the webappclassloader are nullified (see later).

## Webapp class instance as ThreadLocal value

Suppose that we have the following class in the common classpath (for instance in a jar in tomcat/lib) :

```
public class ThreadScopedHolder {
        private final static ThreadLocal<Object> threadLocal = new ThreadLocal<Object>();

        public static void saveInHolder(Object o) {
                threadLocal.set(o);
        }

        public static Object getFromHolder() {
                return threadLocal.get();
        }
}
```

And those 2 classes in the webapp :

```
public class MyCounter {
        private int count = 0;

        public void increment() {
                count++;
        }

        public int getCount() {
                return count;
        }
}
public class LeakingServlet extends HttpServlet {

        protected void doGet(HttpServletRequest request,
                        HttpServletResponse response) throws ServletException, IOException {

                MyCounter counter = (MyCounter)ThreadScopedHolder.getFromHolder();
                if (counter == null) {
                        counter = new MyCounter();
                        ThreadScopedHolder.saveInHolder(counter);
                }

                response.getWriter().println(
                                "The current thread served this servlet " + counter.getCount()
                                                + " times");
                counter.increment();
        }
}
```

If the servlet is invoked at least once, the webapp classloader would not be GCed when the app is stopped: since the classloader of `ThreadScopedHolde` is the common classloader, it remains forever which is as expected. But its `ThreadLocal` instance has a value bound to it (for the non-terminated thread (s) that served the sevlet), which is an instance of a class loaded by the webapp classloader...

Here again, tomcat >=6.0.24 will detect the leak :

```
Mar 17, 2010 10:23:13 PM org.apache.catalina.loader.WebappClassLoader clearThreadLocalMap
SEVERE: A web application created a ThreadLocal with key of type [java.lang.ThreadLocal] (value [java.lang.
ThreadLocal@44676e3f]) and a value of type [test.leak.threadlocal.value.MyCounter] (value [test.leak.
threadlocal.value.MyCounter@62770d2e]) but failed to remove it when the web application was stopped. To prevent
a memory leak, the ThreadLocal has been forcibly removed.
```

## Webapp class instance indirectly held through a ThreadLocal value

Suppose we have the same `ThreadScopedHolder` class (in the common classloader) and `MyCounter` class in the webapp, but with the following servlet :

```
public class LeakingServlet extends HttpServlet {

        protected void doGet(HttpServletRequest request,
                        HttpServletResponse response) throws ServletException, IOException {

                List<MyCounter> counterList = (List<MyCounter>) ThreadScopedHolder
                                .getFromHolder();
                MyCounter counter;
                if (counterList == null) {
                        counter = new MyCounter();
                        ThreadScopedHolder.saveInHolder(Arrays.asList(counter));
                } else {
                        counter = counterList.get(0);
                }

                response.getWriter().println(
                                "The current thread served this servlet " + counter.getCount()
                                                + " times");
                counter.increment();
        }
}
```

We have more or less the same kind of leak as the previous one, but this time tomcat does not detect the leak when stopping the application. The problem is that when it inspects the entries of `ThreadLocalMap`, it checks whether either the key or the value is an instance of a class loaded by the webapp classloader. Here the key is an instance of `ThreadLocal`, and the value is an instance of java.util.`ArrayList`.

The "Find leaks" button in tomcat manager will report the leak when asked :

```
The following web applications were stopped (reloaded, undeployed), but their
classes from previous runs are still loaded in memory, thus causing a memory
leak (use a profiler to confirm):
/testWeb
```

But it does not give any clue about what caused the leak, we would need to make a heapdump and analyse it with some tool like Eclipse MAT.

Tomcat 7.0.6 and later fix this leak by renewing threads in the pool.

## ThreadLocal pseudo-leak

Suppose we have the same `MyCounter` class as above (in the webapp) and the following servlet :

```
public class LeakingServlet extends HttpServlet {
        private ThreadLocal<MyCounter> myThreadLocal = new ThreadLocal<MyCounter>();

        protected void doGet(HttpServletRequest request,
                        HttpServletResponse response) throws ServletException, IOException {

                MyCounter counter = myThreadLocal.get();
                if (counter == null) {
                        counter = new MyCounter();
                        myThreadLocal.set(counter);
                }

                response.getWriter().println(
                                "The current thread served this servlet " + counter.getCount()
                                                + " times");
                counter.increment();
        }

        @Override
        public void destroy() {
                super.destroy();
                // normally not needed, just to make my point
                myThreadLocal = null;
        }
}
```

Notice that the `ThreadLocal` instance is referenced through an instance variable, not a static one.

Sun's implementation of `ThreadLocal` (and `WeakHashMap`) too is such that `ThreadLocalMap` entries whose key is GCed are not immediately removed. (The key is a weak reference to the `ThreadLocal` instance, see `java.lang.ThreadLocal.ThreadLocalMap.Entry<T> in JDK 5/6`. And there's no daemon thread waiting on a `ReferenceQueue`). Instead, it's only during subsequent uses of `ThreadLocal` features that each Thread removes the abandoned `ThreadLocalMap.Entry` entries (see `ThreadLocalMap.expungeStaleEntries()`.

If many threads were used to serve our leaking webapp, but after we stop it only a couple of threads are enough to serve other webapps, one could have some threads that are no longer used, waiting for some work. Since those threads are blocked, they have no interaction with their `ThreadLocalMap` (i.e. there's no `ThreadLocal` value bound to them or removed), so that there's no opportunity to `expungeStaleEntries()`.

Tomcat 6.0.24-6.0.26 "speeds up" the removal of stale entries (and thus fixes the pseudo-leak), by calling `expungeStaleEntries()` for each thread that has some stale entries. Since it's not thread-safe, it has been made optional and disabled by default from 6.0.27.

Tomcat 7.0.6 and later fix the problem by renewing threads in the pool.

## Threads ContextClassLoader

### Threads spawned by webapps

If a webapp creates a thread, by default its context classloader is set to the one of the parent thread (the thread that created the new thread). In a webapp, this parent thread is one of tomcat worker threads, whose context classloader is set to the webapp classloader when it executes webapp code.

Furthermore, the spawned thread may be executing (or blocked in) some code that involves classes loaded by the webapp, thus preventing the webapp classloader from being collected.

So, if the spawned thread is not properly terminated when the application is stopped, the webapp classloader will leak because of the strong reference held by the spawned thread.

Example :

```
public class LeakingServlet extends HttpServlet {
        private Thread leakingThread;

        protected void doGet(HttpServletRequest request,
                        HttpServletResponse response) throws ServletException, IOException {

                if (leakingThread == null) {
                        synchronized (this) {
                                if (leakingThread == null) {
                                        leakingThread = new Thread("leakingThread") {

                                                @Override
                                                public void run() {
                                                        synchronized (this) {
                                                                try {
                                                                        this.wait();
                                                                } catch (InterruptedException e) {
                                                                        e.printStackTrace();
                                                                }
                                                        }
                                                }
                                        };
                                        leakingThread.setDaemon(true);
                                        //leakingThread.setContextClassLoader(null);
                                        leakingThread.start();
                                }
                        }
                }
                response.getWriter().println("Hello world!");
        }
}
```

Here, when the app is stopped, the webapp classloader is still referenced by the spawned thread both through its context classloader and its current call stack (the anonymous Thread subclass is loaded by the webapp classloader).

When stopping an application, tomcat checks the context classloader of every Thread, and if it is the same as the app being stopped, it logs the following message :

```
Mar 18, 2010 11:13:07 PM org.apache.catalina.core.ApplicationContext log
INFO: HTMLManager: stop: Stopping web application at '/testWeb'
Mar 18, 2010 11:13:07 PM org.apache.catalina.loader.WebappClassLoader clearReferencesThreads
SEVERE: A web application appears to have started a thread named [leakingThread] but has failed to stop it.
This is very likely to create a memory leak.
```

Now, if we uncomment the line `leakingThread.setContextClassLoader(null);` in the above example, tomcat (6.0.24) no longer detect the leak when the application is stopped because the spawned thread context classloader is no longer the webapp's. (the "Find leaks" feature in the manager will report it though)

## Threads spawned by classes loaded by the common classloader

Suppose, we have the Commons Pool library in the classpath of the server (e.g. the jar is in tomcat/lib), and the following servlet :

```
public class LeakingServlet extends HttpServlet {
        private GenericObjectPool objectPool;

        protected void doGet(HttpServletRequest request,
                        HttpServletResponse response) throws ServletException, IOException {
                response.getWriter().println("Number of idle objects in the pool :"+objectPool.getNumIdle());
        }

        @Override
        public void init() throws ServletException {
                objectPool = new GenericObjectPool();
                objectPool.setFactory(new BasePoolableObjectFactory() {
                        private AtomicInteger counter = new AtomicInteger(0);

                        @Override
                        public Object makeObject() throws Exception {
                                String str = "Object #" + counter.incrementAndGet();
                                System.out.println("Creating "+str);
                                return str;
                        }

                        @Override
                        public void destroyObject(Object obj) throws Exception {
                                System.out.println("Destroying "+obj);
                        }
                });
                objectPool.setMinIdle(3);
                objectPool.setTimeBetweenEvictionRunsMillis(1000);
        }

        @Override
        public void destroy() {
                try {
                        objectPool.close();
                } catch (Exception e) {
                        e.printStackTrace();
                }
        }
}
```

The call to `GenericObjectPool.setTimeBetweenEvictionRunsMillis)` actually starts or reuses a `java.lang.Timer` shared between all `GenericObjectPool` instances. As long as a pool is running and at least one pool is using the timer eviction feature, the Timer lives.

If there's no other webapp using commons-pool, there's no leak : when we stop the webapp, the servlet is stopped and the pool is closed. If it was the only pool in use, the Timer thread is also stopped and there's no leak.

Now, imagine that there are 2 webapps using commons-pool with the timer eviction feature (imagine the above servlet is deployed in 2 webapps A and B). Suppose webapp A is deployed, then B. Since the commons-pool jar is shared between both webapps, only one Timer thread is spawned, with 2 `TimerTask`, one for each webapp, to handle the eviction of each pool instance.

Then, if we stop webapp A, boom it's leaking! the Timer thread has its context classloader set to the `WebAppClassLoader` of webapp A. This is somehow a bug of commons-pool, but tomcat 6.0.24 tries to help :

```
INFO: HTMLManager: stop: Stopping web application at '/testWeb'
Destroying Object #3
Destroying Object #2
Destroying Object #1
Mar 21, 2010 9:26:36 PM org.apache.catalina.loader.WebappClassLoader clearReferencesStopTimerThread
SEVERE: A web application appears to have started a TimerThread named [Timer-0] via the java.util.Timer API but has failed to stop it. To prevent a memory leak, the timer (and hence the associated thread) has been forcibly cancelled.
```

So the leak is fixed, but unfortunately there's a side effect : it broke webapp B eviction timer. That's why stopping `TimerThread` has been made optional from 6.0.27.

## Threads spawned by JRE classes

Just like third-party libraries may spawn threads that provoke leaks, some JRE classes also spawn threads that inherit from the current class loader and thus provoke leaks.

Instead of trying to stop such threads, tomcat prefers to force the creation of such threads when the container is started, before webapps are started. The `JreMemoryLeakPreventionListener` does it for a few known offenders in the JRE.

## static class variables

When an app is stopped, Tomcat (even before 6.0.24) nullifies the value of all static class variables of classes loaded by the `WebAppClassLoader`. In some cases, it may fix a classloader leak (for example because of a custom `ThreadLocal` class, see above), but even if we still have a leak, it may decrease the amount of memory lost:

Imagine a class with the following variable :

```
private final static byte[] BUFFER = new byte[1024*1024]; //1MB buffer
```

Normally, the 1MB buffer should be freed when the app is stopped, but only if the classloader itself can be garbage-collected. Since there are still possibilities to have a leak of the classloader, clearing the BUFFER variable allows to recover 1MB of memory.

## LogFactory

*to be completed*

## JavaBean Introspector cache

Tomcat calls `java.beans.Introspector.flushCaches();` when an app is stoppped.

## JDBC driver registration

If a webapp contains a JDBC driver (e.g. in WEB-INF/lib), the driver will be registered with the `DriverManager` when it is first used. When the application is stopped, the driver should be deregistered with `DriverManager` to avoid a classloader leak. Since applications usually forget this, tomcat helps by deregistering the driver.

## RMI target

*to be completed*

---

## References

- Mark Thomas interview on DZone
- Eclipse Memory Analysis Tool

Related issues

- 49159 - Improve ThreadLocal memory leak clean-up
- Sun bug 4957990 - In some cases the Server JVM fails to collect classloaders. According to this page it should have been fixed with java 6u16 but actually it was not. It seems to be fixed with 6u21 (documented here and verified by the author of this wiki page).
- Sun bug 6916498 - An exception can keep a classloader in memory if the stack trace that was recorded when it was created contains a reference to one of its classes. Fixes were done in Tomcat for its own classes that had this issue (see BZ 50460), but some library or JRE code may still create a leak that is undetected by tools because of this JVM bug. See also BZ 53936 for a workaround that you can implement if you are unable to fix a buggy library.

---

CategoryFAQ