TESTING THE TO-DO APPLICATION

by

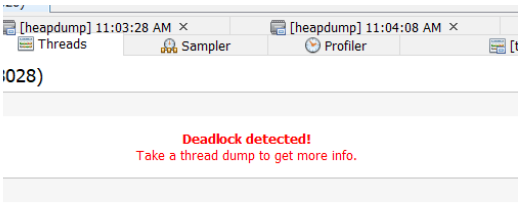
Jean George & Jae Hwang

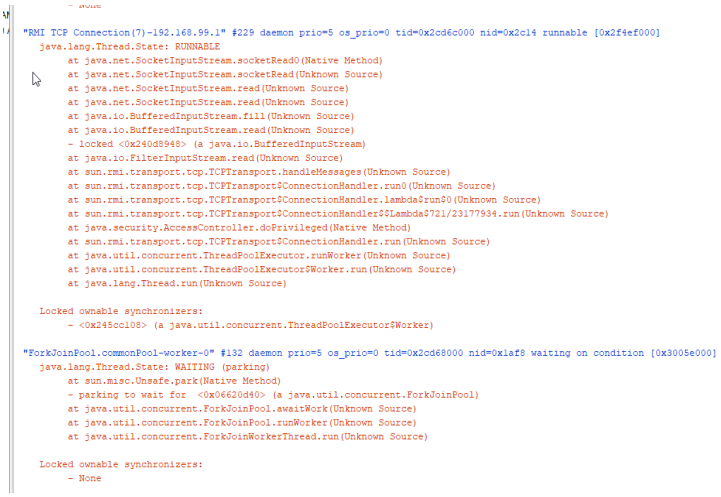
Service Level Agreements:

1. Page Load after Sign In must complete in less than 650ms
2. Completing a Todo must complete in less than 800ms
3. Deleting a Todo must complete in less than 800ms
4. Posting to the Message Board must complete in less than 700ms

1. THREAD DEADLOCK

Hypothesis: On initial assessment, we noticed right off the bat that the JVisualVM was showing a Deadlock Detected warning on the Threads tab. We took a Thread dump and we noticed there was something unusual happening with the *DarkModeServiceImpl.*





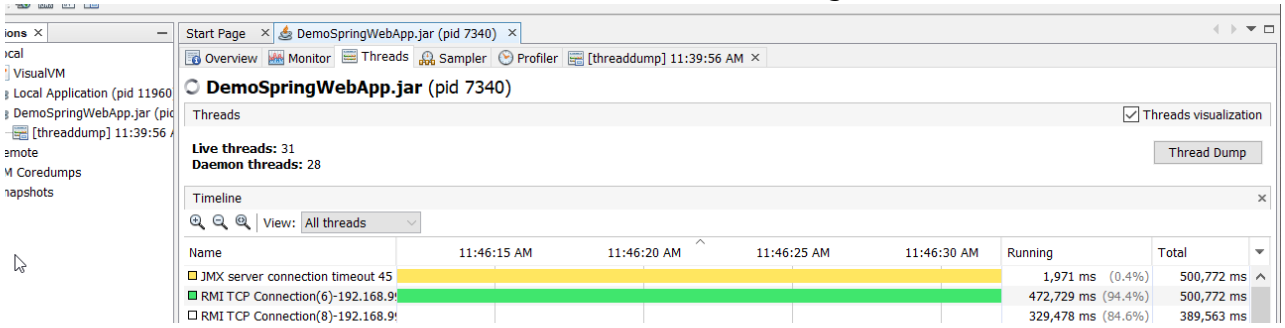
Suggestion: Our suggestion would be to look at the underlying code to look for inefficient implementation or threads that are sharing resources and blocking each other.

Verify:

We looked at the code inside *DarkModeServiceImpl.* We found that the method was spawning synchronized threads that did not necessarily require synchronization and were therefore blocking themselves and causing a deadlock.

We made the changes by removing the code that set synchronization and on rerunning the test, we noticed immediately that there was no more thread deadlock

No more Deadlock Detected Message



2. IDLE ACCUMULATION OF MEMORY

Hypothesis:
We took a look at the Heap while the server was idle and we noticed that over a ten minute period, it accumulated over 200MB. On inspection of the dump, we noticed that there were a large amount of Key-Value pairs being generated.

Suggestion:
Look further into the heap dump and figure out what was causing the issue.

Verify:
On further inspection, we noticed the JNI & MBeans related objects and concluded that it is not part of our AUT.

3. ERRORS & BOTTLENECKS IN TESTS

Hypothesis: We wondered if isolation levels were causing problems between threads because of the massive number of errors and bottlenecks that would occur.

Suggestion: Consider altering the isolation levels around the program and see if it would make a difference

Verify: COULD NOT VERIFY

4. MEMORY LEAK

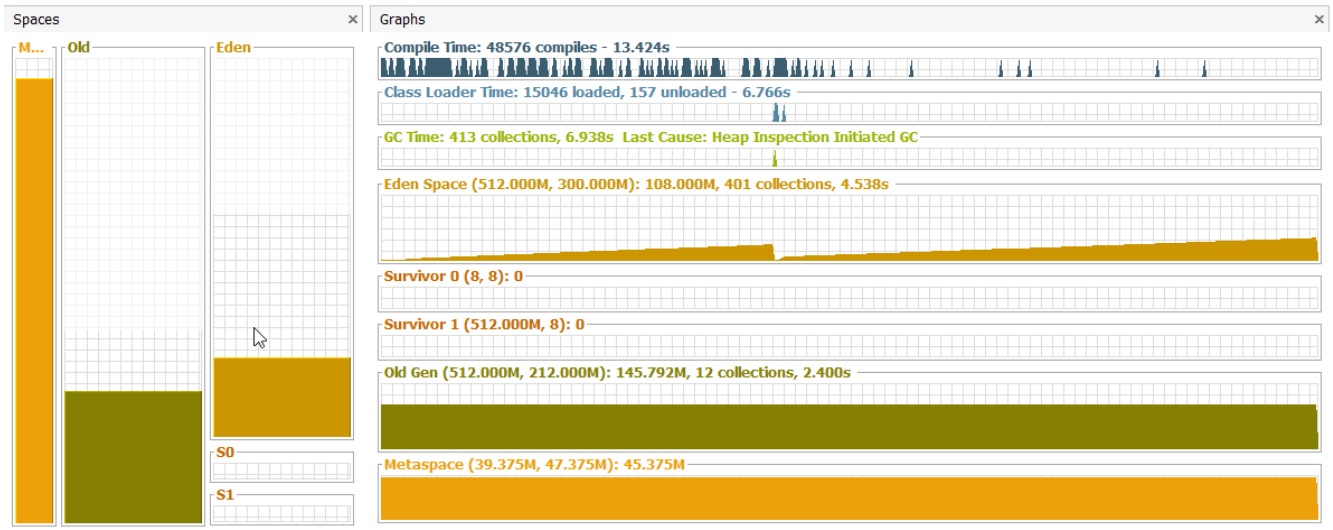
Hypothesis: On running a longer period test, Tenured space does not reduce with full garbage collection but instead keeps on incrementally increasing.

Suggestion: There are objects that are referenced but are no longer being used. We will take a look at the Heap Dump to find out what objects are not being garbage collected from the tenured space and occupying a large amount of the memory.

Verify: On further inspection of the heap dump, we noticed that there were a lot of char[] objects in the Heap that were not being removed even though they were no longer referenced.

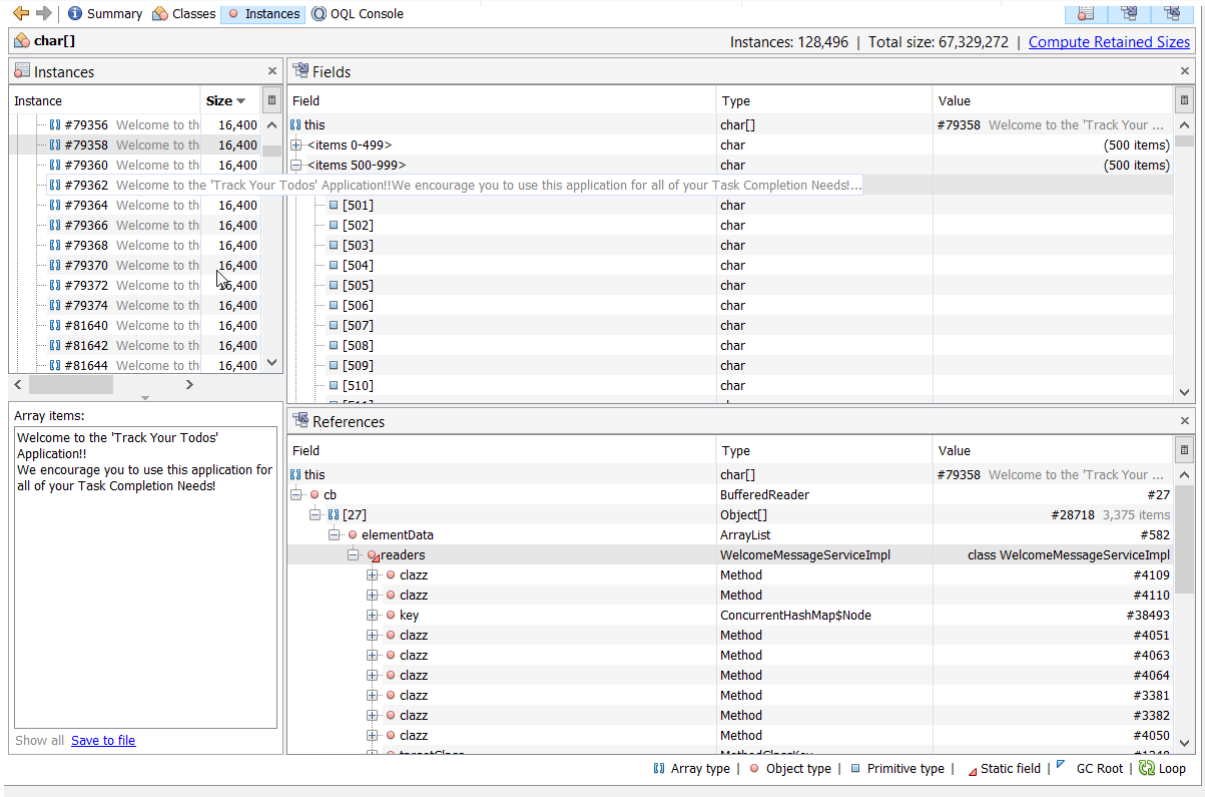
We traced the problem back to the **WelcomeMessageServiceImpl**

•



Visual Stats before implementing change.

Class	Instances	Size	Percentage of Heap
char[]	128,022	63.80 MB	49.15%
byte[]	6,526	26.74 MB	20.60%
org.h2.value.Value[]	160,090	18.45 MB	14.21%
java.lang.Object[]	36,730	10.38 MB	7.99%
java.lang.String	118,966	1.82 MB	1.40%
java.lang.Class	16,016	1.45 MB	1.11%
java.util.TreeMap$Entry	44,450	1.36 MB	1.04%
java.util.concurrent.ConcurrentHashMap$Node	53,355	1.22 MB	0.94%
java.lang.reflect.Method	14,106	1.18 MB	0.91%
com.revature.model.Message	49,864	1.14 MB	0.88%



In the WelcomeMessageServiceImpl file, the Buffered reader was being added to the '**readers'** array, even though it is only used once and never referenced again.

We refactored the code to not store in '**readers'** array by deleting it. We repackaged and ran the application and we immediately noticed efficient Garbage collection and significantly reduced size of the Tenured Space.