# Profiling of OKE Services

## INTRODUCTION

Profiling is the process of examining an application to locate memory or performance-related issues. When profiling a Java application, you can monitor the Java Virtual Machine (JVM) and obtain data about application performance, object allocation, garbage collection and many more.

VisualVM/jConsole are powerful profiling tools which provide a visual interface to see deep and detailed information about local and remote Java applications while they are running on a Java Virtual Machine (JVM). They help the programmers and architects to track memory leaks, analyze the heap data, monitor the garbage collector and CPU profiling.

## STEPS TO ENABLE PROFILING:

### LOCAL RUNNING APPS:

There is no extra configuration required to profile local JVM applications. VisualVM/jConsole can automatically detect and connect to JVM applications that are running on local machine and shows it in the Applications tab.

### OKE(REMOTE) APPLICATIONS:

We can connect to remote JVM applications with the help of JMX technology by simply following the below steps.

1. Restart the OKE service by adding the following system properties.
   **1.Deployment.yaml:**
   ....
   ```
   - name: JAVA_OPTS
     value: "-Dcom.sun.management.jmxremote=true \
             -Dcom.sun.management.jmxremote.authenticate=false \
             -Dcom.sun.management.jmxremote.ssl=false \
             -Dcom.sun.management.jmxremote.local.only=false \
             -Dcom.sun.management.jmxremote.port=9010 \
             -Dcom.sun.management.jmxremote.rmi.port=9010 \
             -Djava.rmi.server.hostname=127.0.0.1"
   ```
   ....
   **2.Service.yaml:**
   ports:
   ....
   ```
   - name: jmx
     protocol: TCP
     port: 9010
   ```
   ....
   **3.Dockerfile:**
   ENTRYPOINT java $JVM_OPTS $JAVA_OPTS -cp app:app/lib/* oal.oracle.apps.spm.... (Add $JAVA_OPTS to the existing ENTRYPOINT)
2. Once you restart the service and able to expose JMX port, do Local Port Forward on jmx port(9010 in this case)
   eg: kubectl port-forward [podname] 9010:9010 -n [namespace] --kubeconfig=kubeconfig-dev
3. In VisualVM, Application should be visible in Applications section, if not, right click on Local and Add JMX connection.
4. In terminal, type jconsole and console window will pop up. Under Remote Process, Add localhost:[jmxport](9010 in this case).

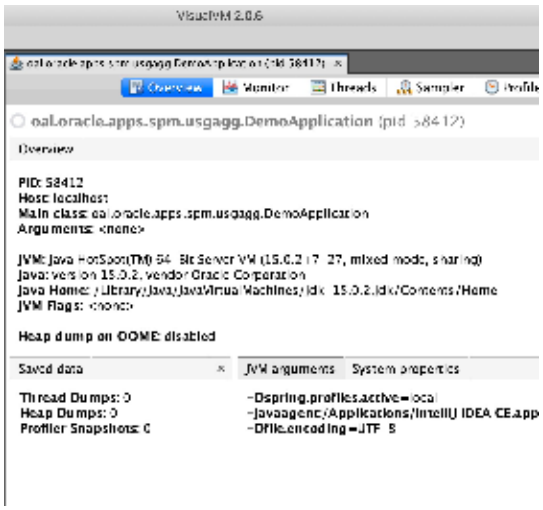
## PROFILING THE APPLICATION:

### With VisualVM:

Once you double click on the Application, you would be seeing below tabs.

#### OVERVIEW:

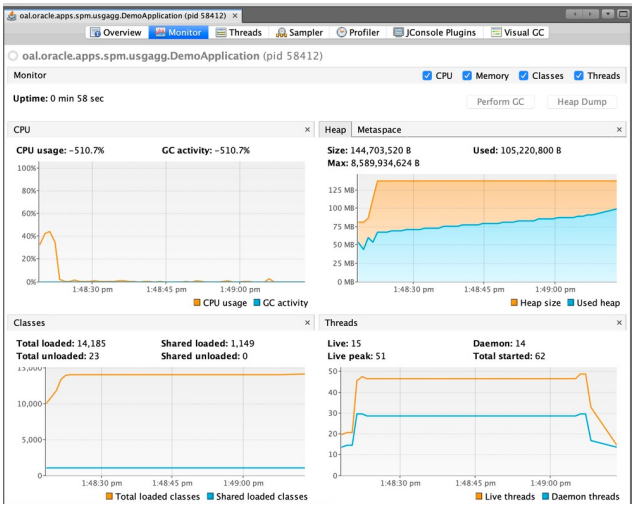
In the Overview tab, all the static information about the application is displayed. You can see the PID,Host,System Properties etc.,




**MONITOR:**

In the Monitor tab, you will be able to see uptime of the application and information about CPU, Memory, Classes and Threads. You can customise the tabs to hide the graphs by unselecting in the top right corner.

You can take Heap Dump and Perform manual GC by clicking on buttons provided above the graphs. Taking multiple Heap dumps helps us in understanding if there are any memory leaks in the application.



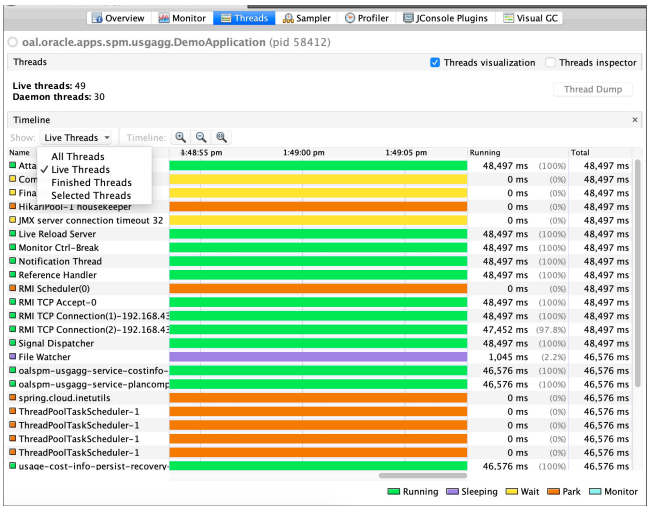

**THREADS:**

Here, you will be able to get information on state of the application threads. By default, it shows about Live Threads, but you can select All,Finished, Live threads from the drop down list.

From here, you can take Thread Dump to understand the application. Once you take a Thread Dump, it appears in the Applications tab window under the name of the Application.You can analyse different timed Thread Dumps to understand if there is an issue with threads in the application.

Also, In the bottom right corner, you are provided with color coding information to understand the state of the threads.

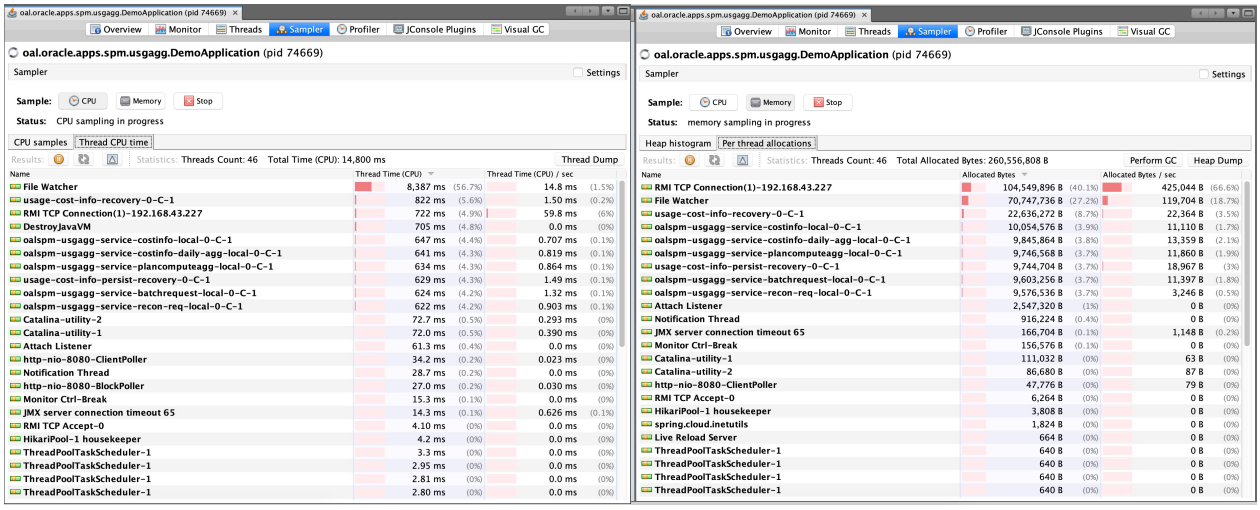
You can add a plugin Thread inspector, if you want to get stack trace of any particular thread.




## SAMPLER:

In CPU Sampling, you can view detailed data on method-level CPU performance (execution time), showing the total execution time and total CPU time. When analyzing application performance, Java VisualVM instruments all of the methods of the profiled application.

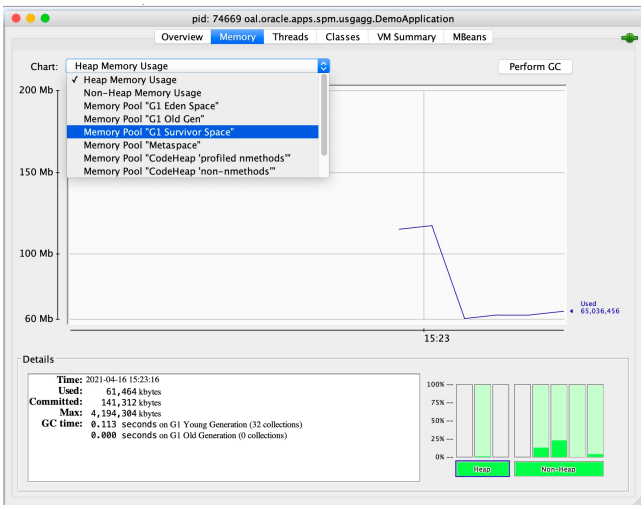
In Memory Sampling, JVisualVM displays the total number of objects allocated by each class (including array classes) in a table. For each class currently loaded class in the Java Virtual Machine (JVM), the profiling results display the size and number of objects allocated since the profiling session started. The results are automatically updated as new objects are allocated and as new classes are loaded.




## WITH JCONSOLE:

## MEMORY:

This section is very useful to understand the memory foot-print of the application. It shows percentage of all the memory pools in HEAP as well as NON-HEAP.




## Non-heap memory

### Problem statement

1. Application consumes much more memory than the amount specified via parameters: Xms, Xmx, ...
2. Kubernetes kills the pods with OOMKilled message while Heap and Metaspace consumption is normal
3. VisualVM and Jprofiler shows normal cpu and memory usage
   There must be a problem with the unmonitored, off heap memory...

### Native memory tracking

Startup jvm param: -XX:NativeMemoryTracking

Monitoring on OKE:

Login to pod:

```
# kubectl -n [namespace] exec -i -t [podname] bash
```

Navigate to jcmd:

```
# cd /usr/java/jdk-11.0.3/bin/
```

Get native memory tracking result:

```
# ./jcmd 1 VM.native_memory
```

To get live data you can use watch command: # watch ./jcmd 1 VM.native_memory

### Non heap components

Limitable (by jvm options)

1. Code Cache  XX:ReservedCodeCacheSize  compiled code
2. Metaspace  XX:MaxMetaspaceSize  Loaded class metadata
3. Symbol  XX:StringTableSize  Stringpool

Unlimitable

1. Thread stack  Thread stack size can be controlled Xss
2. Compiler  JIT compiler used memory to generate code

## Just in Time compiler

A JIT compiler compiles byte code to native code for frequently executed sections. There are two types of JIT compilers available in JVM.

1. C1 - Client Compiler
2. C2 - Server Compiler

### Client compiler (C1)

Optimized for faster start-up time but less optimized code.
Most of the application starts faster.
Generally used for short-lived applications.

### Server compiler (C2)

C2 observes and analyzes the code over a longer period of time compared to C1. Hence slower but more optimized code.
Used for long-running server-side applications.
Better runtime performance after steady-state.

### Tiered Compilation

Goal is to use a mix of C1 and C2 compilers in order to achieve both fast startup and good long-term performance. First C1 is used to compile, while the compiler profiling is running, it uses C2 to compile it again for faster execution (2 times compiling)
This can be controlled by adding -XX:-TieredCompilation to JVM_OPTS in deployment.yaml which disables intermediate compilation.