How to profile AeroWebb endpoint

How to profile AeroWebb endpoint

Prerequisites

- Have a working AeroWebb installation see developer guide.
- Enable JFR for tomcat
- Install JMC
- Optional : install VisualVM.
- Optional : increase the stack depth limit for JFR stack traces (see JFR config page for more info)

```
set CATALINA_OPTS=-XX:+FlightRecorder -XX:FlightRecorderOptions=stackdepth=1024
```

Profiling

Definition from wikipedia.org :

> In software engineering, profiling ("program profiling", "software profiling") is a form of dynamic program analysis that measures, for example, the space (memory) or time complexity of a program, the usage of particular instructions, or the frequency and duration of function calls. Most commonly, profiling information serves to aid program optimization, and more specifically, performance engineering.
>
> Profiling is achieved by instrumenting either the program source code or its binary executable form using a tool called a profiler (or code profiler). Profilers may use a number of different techniques, such as event-based, statistical, instrumented, and simulation methods.

In this section we will see how toprofile and analyze an AeroWebb endpoint.

Start Tomcat

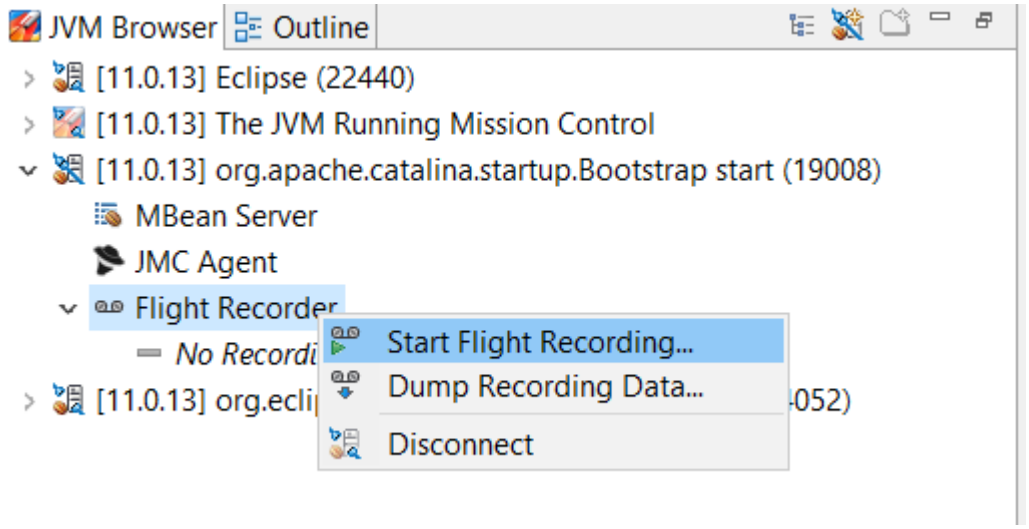
Start tomcat with the aerowebb.war version you want to profile. Ensure JFR has been enabled before startup.

Target the endpoint(s) you want to profile

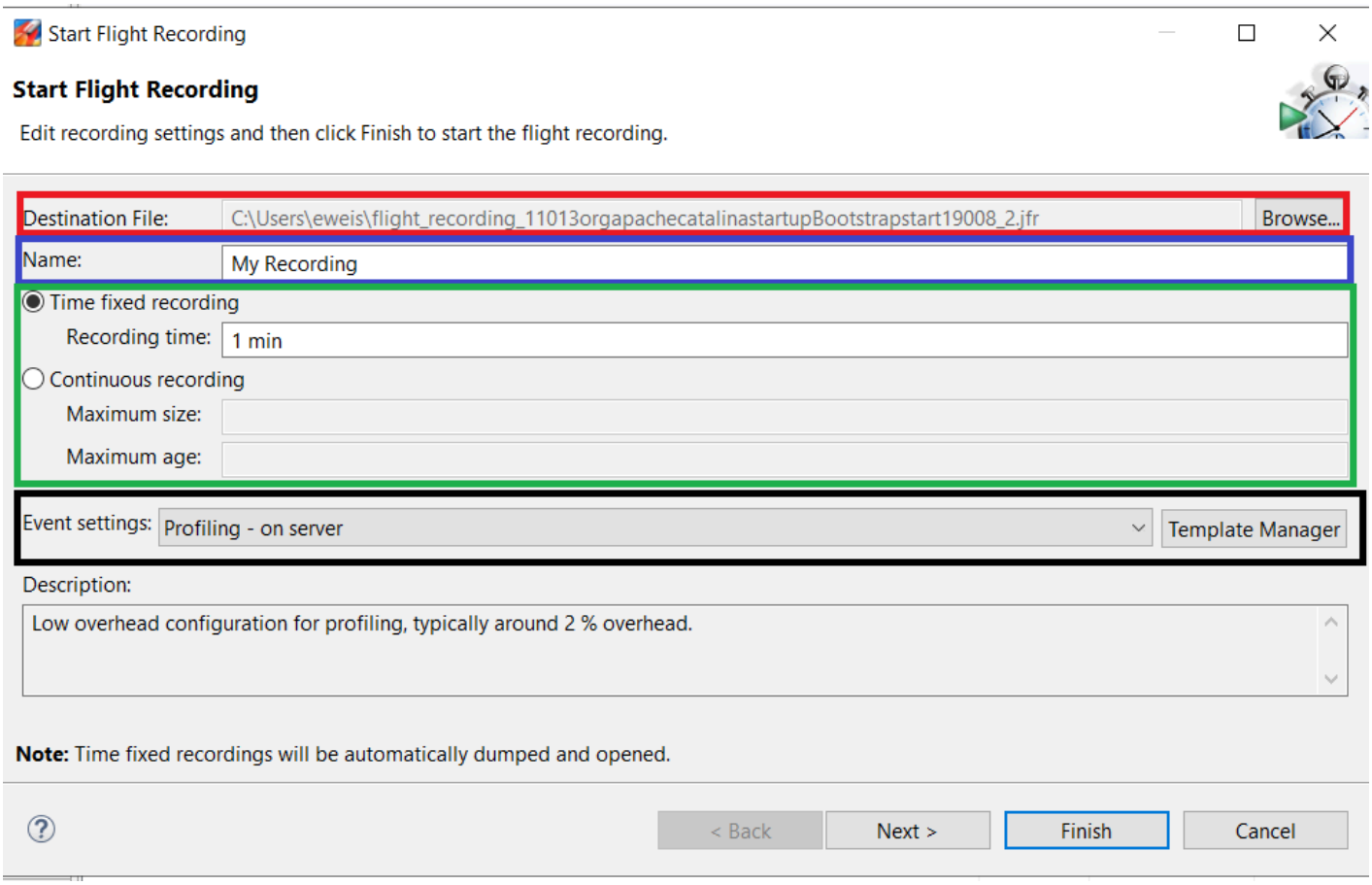
To avoid recording superfluous data and precisely target the methods you want to analyze, preemptively take any steps such as login into AeroWebb (and getting to the screen/pop-up/button/etc you want to analyze if using the AeroWebb GUI).

Start a recording

- Start JMC and find your tomcat process in the JVM Browser
- Right click on the Flight Recorder button and select *Start Flight Recording*



A wizard opens to help you configure the recording :



In descending order :

1. In red : customize the output folder where the .jfr file will be saved
2. In blue : customize the name of the recording
3. In green : customize the length of the recording. If you don't know how long the endpoint(s) you want to profile will take, you can choose continuous. Either way you can stop the recording at any time.
4. In black : customize the template you want to use for the recording settings. Dictates how and which events are recorded.

By default, JMC comes with 2 base templates : Profiling (on server) and Continuous (on server).

Continuous has less overhead and is designed to be safe to use in production environments while Profiling is more for in-depth analysis (recommended setting).

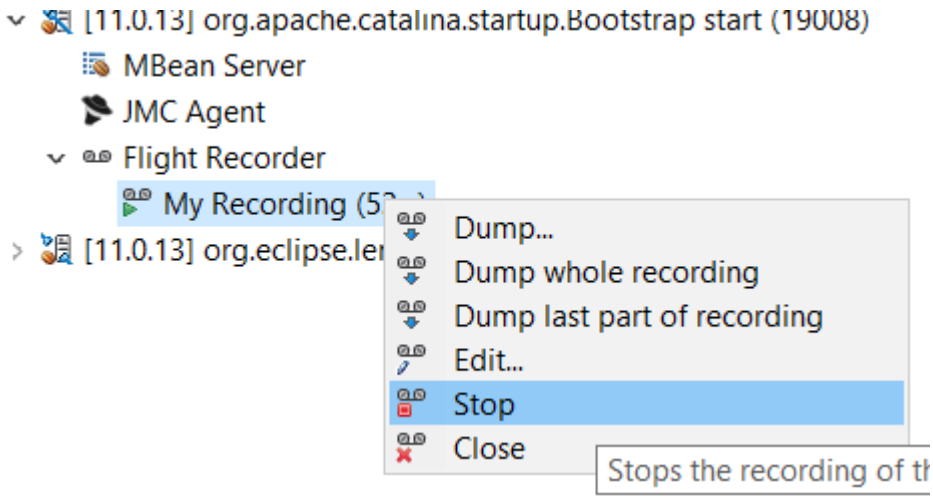
If you need to fine-tune more settings, you can check what is available in *Next*. Default configuration should be enough in most cases.

Click on finish to start the recording.

Collect data

Once the recording has started, call the endpoint(s) you want to analyze (from AeroWebb GUI, Postman etc.). During the duration of the recording, all events will be recorded. Once the recording has ended, a .jfr file containing all information will be generated and written to disk.

If you specified a duration, you can let the recording run out. But again, in order to minimize noise in the collected data, it is suggested to manually stop the recording once your endpoint(s) are done processing. To do so, right click on the recording and select *Stop*.

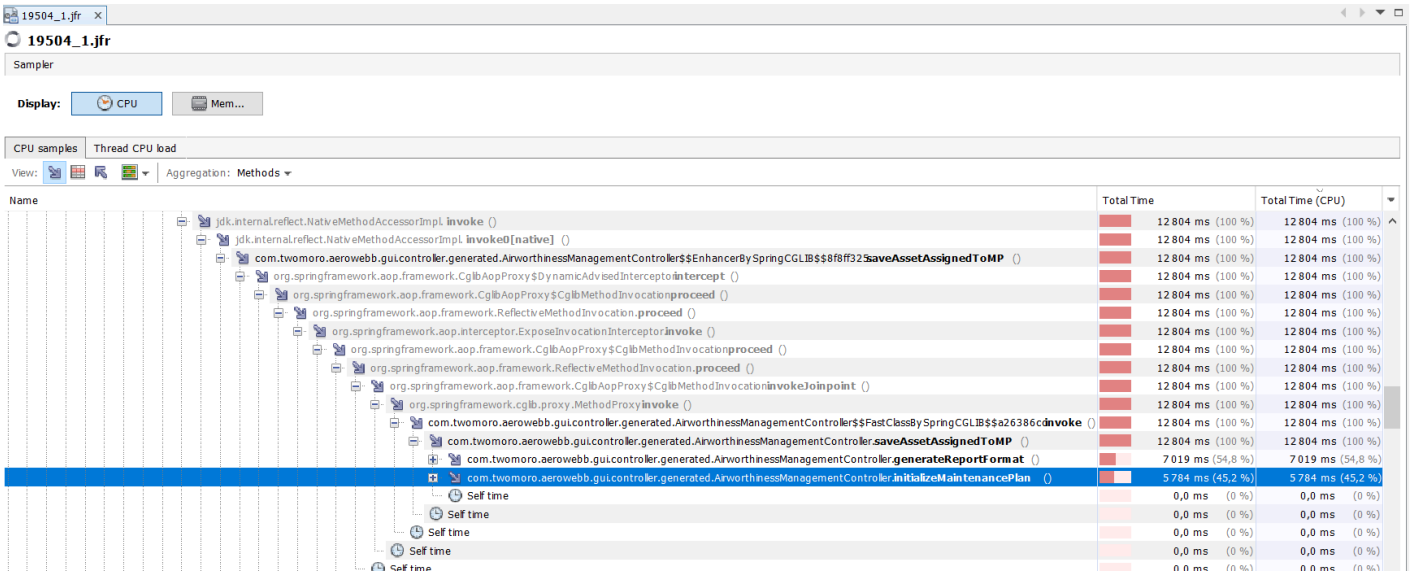


Once the recording has ended, the .jfr file is generated and saved to disk and a view is opened in JMC with the results.

Timing of methods

JMC offers a rich summary of what is inside the .jfr file, but it might not be enough : for instance, I have yet to figure out how to display time spent in each methods.

Fortunately, VisualVM has this feature and can open .jfr files. Head to Sampler → CPU to display this information. In some cases, it might be tricky to find the thread on which your methods executed but ordering by *Total Time (CPU)* usually helps. Also, some time values might be deceiving and not reflect what really happens, so caution is advised.


19504_1.jfr
19504_1.jfr
Sampler
Display:
CPU
Mem...
CPU samples
Thread CPU load
View:
Aggregation: Methods
Name
Total Time
Total Time (CPU)
jdk.internal.reflect.NativeMethodAccessorImpl.invoke ()
12 804 ms (100 %)
jdk.internal.reflect.NativeMethodAccessorImpl.invoke0[native] ()
com.twomoro.aerowebb.gui.controller.generated.AirworthinessManagementController$$EnhancerBySpringCGLIB$$8f8ff325saveAssetAssignedToMP ()
org.springframework.aop.framework.CglibAopProxy$DynamicAdvisedInterceptorintercept ()
org.springframework.aop.framework.CglibAopProxy$CglibMethodInvocationproceed ()
org.springframework.aop.framework.ReflectiveMethodInvocation.proceed ()
org.springframework.aop.interceptor.ExposeInvocationInterceptorinvoke ()
org.springframework.aop.framework.CglibAopProxy$CglibMethodInvocationproceed ()
org.springframework.aop.framework.ReflectiveMethodInvocation.proceed ()
org.springframework.aop.framework.CglibAopProxy$CglibMethodInvocationinvokeJoinpoint ()
org.springframework.cglib.proxy.MethodProxyinvoke ()
com.twomoro.aerowebb.gui.controller.generated.AirworthinessManagementController$$FastClassBySpringCGLIB$$a26386cdinvoke ()
com.twomoro.aerowebb.gui.controller.generated.AirworthinessManagementController.saveAssetAssignedToMP ()
com.twomoro.aerowebb.gui.controller.generated.AirworthinessManagementController.generateReportFormat ()
7 019 ms (54,8 %)
com.twomoro.aerowebb.gui.controller.generated.AirworthinessManagementController.initializeMaintenancePlan ()
5 784 ms (45,2 %)
Self time
0,0 ms (0 %)