

# Quarkus - Measuring Performance

This guide covers:

- how we measure memory usage
- how we measure startup time
- which additional flags will Quarkus apply to native-image by default

All of our tests are run on the same hardware for a given batch. It goes without saying but it's better when you say it.

## How do we measure memory usage

When measuring the footprint of a Quarkus application, we measure [Resident Set Size \(RSS\)](#) and not the JVM heap size which is only a small part of the overall problem. The JVM not only allocates native memory for heap (`-Xms`, `-Xmx`) but also structures required by the jvm to run your application. Depending on the JVM implementation, the total memory allocated for an application will include, but not limited to:

- Heap space
- Class metadata
- Thread stacks
- Compiled code
- Garbage collection

## Native Memory Tracking

In order to view the native memory used by the JVM, you can enable the [Native Memory Tracking \(NMT\)](#) feature in hotspot;

Enable NMT on the command line;

```
-XX:NativeMemoryTracking=[off | summary | detail] ①
```

① NOTE: this feature will add cause an approximately 5-10% performance overhead

It is then possible to use `jcmd` to dump a report of the native memory usage of the Hotspot JVM running your application;

```
jcmd <pid> VM.native_memory [summary | detail | baseline |  
summary.diff | detail.diff | shutdown] [scale= KB | MB | GB]
```

# Cloud Native Memory Limits

It is important to measure the whole memory to see the impact of a Cloud Native application. It is particularly true of container environments which will kill a process based on its full RSS memory usage.

Likewise, don't fall into the trap of only measuring private memory which is what the process uses that is not shareable with other processes. While private memory might be useful in a environment deploying many different applications (and thus sharing memory a lot), it is very misleading in environments like Kubernetes/OpenShift.

## Platform Specific Memory Reporting

In order to not incur the performance overhead of running with NVM enabled, we measure the total RSS of an JVM application using tools specific to each platform.

### Linux

The linux [pmap](#) and [ps](#) tools provide a report on the native memory map for a process

```
$ ps -o pid,rss,command -p <pid>
```

```
PID    RSS COMMAND
11229 12628 ./target/getting-started-1.0-SNAPSHOT-runner
```

```
$ pmap -x <pid>
```

```
13150:  /data/quarkus-application -Xmx100m -Xmn70m
Address          Kbytes      RSS      Dirty Mode  Mapping
0000000000400000    55652    30592         0 r-x-- quarkus-application
00000000003c58000         4         4         4 r-x-- quarkus-application
00000000003c59000     5192     4628     748 rwx-- quarkus-application
000000000054c0000      912     156     156 rwx-- [ anon ]
...
00007fcd13400000     1024     1024    1024 rwx-- [ anon ]
...
00007fcd13952000         8         4         0 r-x-- libfreebl3.so
...
-----
total kB          9726508  256092  220900
```

Each Memory region that has been allocated for the process is listed;

- Address: Start address of virtual address space
- Kbytes: Size (kilobytes) of virtual address space reserved for region
- RSS: Resident set size (kilobytes). This is the measure of how much memory space is actually

being used

- Dirty: dirty pages (both shared and private) in kilobytes
- Mode: Access mode for memory region
- Mapping: Includes application regions and Shared Object (.so) mappings for process

The Total RSS (kB) line reports the total native memory the process is using.

## macOS

On macOS, you can use `ps x -o pid,rss,command -p <PID>` which list the RSS for a given process in KB (1024 bytes).

```
$ ps x -o pid,rss,command -p 57160

PID      RSS COMMAND
57160 288548 /Applications/IntelliJ IDEA
CE.app/Contents/jdk/Contents/Home/jre/bin/java
```

Which means IntelliJ IDEA consumes 281,8 MB of resident memory.

## How do we measure startup time

Some frameworks use aggressive lazy initialization techniques. It is important to measure the startup time to first request to most accurately reflect how long a framework needs to start. Otherwise, you will miss the time the framework *actually* takes to initialize.

Here is how we measure startup time in our tests.

We create a sample application that logs timestamps for certain points in the application lifecycle.

```

@Path("/")
public class GreetingEndpoint {

    private static final String template = "Hello, %s!";

    @GET
    @Path("/greeting")
    @Produces("application/json")
    public Greeting greeting(@QueryParam("name") String name) {
        System.out.println(new SimpleDateFormat("HH:mm:ss.SSS")
            .format(new java.util.Date(System.currentTimeMillis())));
        String suffix = name != null ? name : "World";
        return new Greeting(String.format(template, suffix));
    }

    void onStart(@Observes StartupEvent startup) {
        System.out.println(new SimpleDateFormat("HH:mm:ss.SSS")
            .format(new Date()));
    }
}

```

We start looping in a shell, sending requests to the rest endpoint of the sample application we are testing.

```

$ while [[ "$(curl -s -o /dev/null -w '%{http_code}''
localhost:8080/api/greeting)" != "200" ]]; do sleep .00001; done

```

In a separate terminal, we start the timing application that we are testing, printing the time the application starts

```

$ date +"%T.%3N" && ./target/quarkus-timing-runner

10:57:32.508
10:57:32.512
2019-04-05 10:57:32,512 INFO [io.quarkus] (main) Quarkus 0.11.0
started in 0.002s. Listening on: http://127.0.0.1:8080
2019-04-05 10:57:32,512 INFO [io.quarkus] (main) Installed
features: [cdi, resteasy, resteasy-jsonb]
10:57:32.537

```

The difference between the final timestamp and the first timestamp is the total startup time for the application to serve the first request.

# Additional flags applied by Quarkus

When Quarkus invokes GraalVM `native-image` it will apply some additional flags by default.

You might want to know about the following ones in case you're comparing performance properties with other builds.

## Disable fallback images

Fallback native images are a feature of GraalVM to "fall back" to run your application in the normal JVM, should the compilation to native code fail for some reason.

Quarkus disables this feature by setting `-H:FallbackThreshold=0`: this will ensure you get a compilation failure rather risking to not notice that the application is unable to really run in native mode.

If you instead want to just run in Java mode, that's totally possible: just skip the native-image build and run it as a jar.

## Disable Isolates

Isolates are a neat feature of GraalVM, but Quarkus isn't using them at this stage.

Disable via `-H:-SpawnIsolates`.

## Disable auto-registration of all Service Loader implementations

Quarkus extensions can automatically pick the right services they need, while GraalVM's native-image defaults to include all services it's able to find on the classpath.

We prefer listing services explicitly as it produces better optimised binaries. Disable it as well by setting `-H:-UseServiceLoaderFeature`.

## Better default for Garbage Collection implementation

The default in GraalVM seems meant to optimise for short lived processes.

Quarkus defaults to server applications, so we switch to a better default by setting `-H:InitialCollectionPolicy=com.oracle.svm.core.genscavenge.CollectionPolicy$BySpaceAndTime`.

## Others ...

This section is provided as high level guidance, but can't presume to be comprehensive as some flags are controlled dynamically by the extensions, the platform you're building on, configuration details, your code and possibly a combination of any of these.

Generally speaking the ones listed here are those most likely to affect performance metrics, but in the right circumstances one could observe non negligible impact from the other flags too.

If you're to investigate some differences in detail make sure to check what Quarkus is invoking exactly: when the build plugin is producing a native image, the full command lines are logged.