GRAALVM

Java frameworks for the cloud: Establishing the bounds for rapid startups

Making the case for using the GraalVM platform for serverless applications

by Oleg Šelajev

August 3, 2020

Download a PDF of this article

The landscape of Java frameworks is vast and full of excellence. You can find one for any taste: Java EE–based, Spring-based, cloud native, Kubernetes native, flexible or opinionated, full-stack or modular, as well as ones that are lightweight, dev-friendly, operations-friendly, DevOps-friendly, cloud-friendly, and MicroProfile-based. You can find microframeworks, frameworks that work as an integration point of the ecosystem, and frameworks you use as libraries and integrate with other libraries yourself, as well as those that are modern, performant, easy-to-use, Kotlin-ready, expressive and elegant, or any other combination of positive adjectives in the vocabulary of modern framework authors.

All of those frameworks offer at least the basic functionality needed to create a web application that can respond to HTTP requests. How do you pick the best one? Practical wisdom suggests you should stick to the framework your team knows best and one that is popular enough that it's easy to find other people who know it sufficiently well.

This boring, rational explanation does have a seed of engineering truth to it. Most frameworks, at least the popular ones, offer similar functionality, are based on similar lower-level stacks (often Netty), support similar annotations and

programming models, offer built-in dependency injection mechanisms, and so on.

They all look a bit similar—and while there are differences, many of them are “soft” and a preference for them can be explained as “taste.” On the other hand, some of the differences affect the idiomatic choices and performance of real-world applications.

Exploring the cloud options

Workloads in the cloud run on other people’s servers, and you usually pay for consumed resources as you go. In addition, the performance of your code directly affects the cost of running your software in the cloud. Slow means expensive. Fast means cheaper.

The problem with this seemingly straightforward reasoning is that *good performance* and *worse performance* mean different things for different applications; sometimes *good performance* means good throughput, sometimes it means low memory usage, and sometimes it means a fast cold-start startup time.

Depending on what’s most important for your particular use case, you’d have a different perspective on the importance of a given performance metric. Even without changing the framework, you can get a performance profile that is better suited to your use case if you run your application appropriately. For example, you might need to tune the runtime parameters you’re passing to the Java process, such as figuring out a better garbage collection configuration or picking a different distribution that could run your applications faster.

Figure 1 is a handy diagram that explains how you would want to run your Java application to achieve the performance profile you want. If you want the best throughput for your code—that is, you want to serve as many user requests as possible using the same resources—consider running your application with Oracle GraalVM Enterprise Edition (GraalVM Enterprise). This article makes the case for using it, particularly for serverless applications. I’d argue that the just-in-time (JIT) compiler in GraalVM Enterprise is probably the most powerful of those currently available, and very often it offers superior throughput. If you are interested in optimizing tail latencies and making sure your application won’t get stuck for periods of time, then either you should have the best JIT compiler available or you need to pick a low-latency GC algorithm.

For many cloud workloads, low memory usage and fast startup are the most essential metrics. For those, GraalVM Native Image offers superior runtime performance. Native Image allows you to compile your application ahead of time to a native binary, which provides cold startup times in range of a few dozen milliseconds and lower memory usage due to not needing to include and use any JIT compilation facilities.

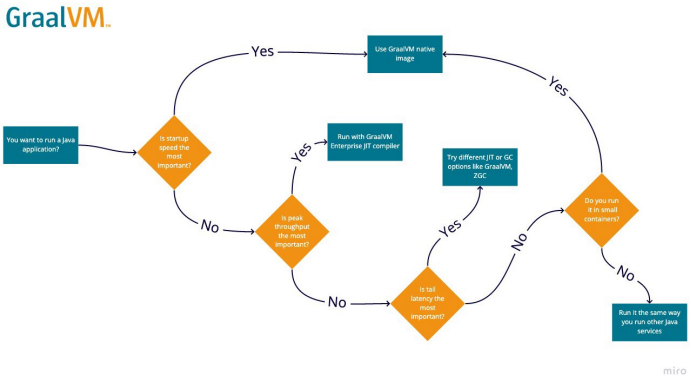

Figure 1. How to run a Java application to achieve a desired performance profile (view the full-size image)

Determining a baseline for Java apps

I used a few frameworks to determine a baseline performance profile for a Java application in a cloud setting by asking

- How fast can the application start?
- How soon can the application do some useful work?
- How much memory does the application use?

To find the answers, the GraalVM team at Oracle Labs ran a few small "helloworld" applications I built with various frameworks, which I then ran in a small, resource-constrained Docker container—a scenario that resembles the ephemeral setup of cloud deployments.

For this project, I configured the Docker containers with 128 MB of memory. This was not an accidental, arbitrary number: Many serverless solutions have billing that is tied to consumption based on gigabytes of memory per second, and the first step is a 128 MB environment. Thus, these experiments approximated the performance you could see in the smallest and cheapest environments—with a specific focus on startup time, which is critical for serverless functions.

Here are the results for the Dropwizard, Micronaut, and Quarkus frameworks, along with the instructions I used to build a "helloworld" app to meet the test requirements.

Dropwizard. By following the official Dropwizard getting started guide, it was easy to start the Maven archetype and build the getting-started code. Here are the steps:

1. Create the project by running the following command:

```
mvn archetype:generate -DarchetypeGroupId=
```

2. Respond to the interactive questions:

```
make 'artifactId: dropwizard-getting-starte
```

3. Change directories:

```
cd dropwizard-getting-started
```

4. Run the following command:

```
mvn verify
```

5. Create a simple Dockerfile:

```
FROM adoptopenjdk/openjdk11:alpine-slim
COPY target/dropwizard-getting-started-1.0-
EXPOSE 8080
CMD java -jar complete.jar server
```

6. Build the Docker image:

```
docker build -f Dockerfile -t shelajev/drop
```

7. Run the container, giving it 128 MB of memory:

```
docker run --rm --memory 128M -p 8083:8080
```

When I ran the app, it self-reported that the server was started in 3.5 seconds:

```
INFO  [2020-06-16 22:16:38,386] org.eclipse.j
```

This is a decent result. It's not ideal for serverless applications, because that is 3.5 seconds you'd need to pay for while your application was not actually doing anything useful. But this was a very basic test, and the framework was not tuned for a rapid start. On the other hand, a larger application using any framework would tend to start slower due to the need to load additional classes, initialize some caches, hydrate class hierarchies or plugin systems, and perform other tasks at the beginning of the application runtime.

Micronaut. Getting started with Micronaut was super easy. I followed the Micronaut getting started guide, and in no time I had the application ready.

For this test, I used Micronaut 1.3.6, which was the latest release at the time of writing. Here are the commands I used. They are very similar to what you could expect from a getting started guide.

```
mn create-app example.micronaut.complete
mv complete micronaut-getting-started
cd micronaut-getting-started
./gradlew build
docker build -f Dockerfile -t shelajev/micron
```

Note that the default Dockerfile is interesting: It uses an Eclipse OpenJ9–based base image, which some developers think is a great choice for the runtime. And it provides some configuration, specifying the heap size and the tuning options.

Here are the commands I used:

```
FROM adoptopenjdk/openjdk13-openj9:jdk-13.0.2
COPY build/libs/complete-*-all.jar complete.j
EXPOSE 8080
CMD ["java", "-Dcom.sun.management.jmxremote"
```

When I ran the app, it self-reported a startup time of 1.6 seconds:

```
> docker run --rm --memory 128M -p 8083:8080
23:02:02.716 [main] INFO  io.micronaut.runtim
```

Quarkus. Quarkus is self-described as a “Kubernetes Native Java stack.” (The project’s website actually says, “Quarkus provides a cohesive, fun to use, full-stack framework by leveraging a growing list of over fifty best-of-breed libraries that you love and use.”)

I ran the same experiment with Quarkus and checked the results. Here is the Quarkus guide I used, which is similar to the other guides.

I ran the following commands to build a Docker image (in this case, using a Maven command):

```
mvn io.quarkus:quarkus-maven-plugin:1.5.1.Fin
    -DprojectGroupId=org.acme \
    -DprojectArtifactId=getting-started \
    -DclassName="org.acme.getting.started.Gre
    -Dpath="/hello"
cd getting-started
./mvnw quarkus:add-extension -Dextensions="co
./mvnw clean package -Dquarkus.container-imag
```

When I ran the Quarkus application, the application started in a respectable 2 seconds:

```
> docker run --rm --memory 128M -p 8083:8080
exec java -Dquarkus.http.host=0.0.0.0 -Djava.
__  ____  __  _____   ___  __ ____  ______
 --/ __ \/ / / / _ | / _ \/ //_/ / / / __/
 -/ /_/ / /_/ / __ |/ , _/ ,< / /_/ /\ \
--\___\_\____/_/ |_/_/|_/_/|_|\____/___/
2020-06-16 23:16:14,161 INFO  [io.quarkus] (m
2020-06-16 23:16:14,217 INFO  [io.quarkus] (m
2020-06-16 23:16:14,218 INFO  [io.quarkus] (m
```

Now you are probably wondering whether Micronaut is the best. After all, the tests above started in a few seconds with Micronaut leading by a bit. It might be the best. However, the real point of running these different frameworks in a very simple “getting-started helloworld” setting was to establish a baseline to compare against running those applications using GraalVM Native Image.

Enter GraalVM Native Image

At least two of the frameworks in this article work well with the GraalVM Native Image feature. So I tried to establish another bound: this time, sort of an upper bound on the startup time in a very constrained environment similar to the environment you would want to use in the cloud.

I performed the same test with a sample Spring framework application using the work-in-progress Spring-GraalVM-native feature. This project is currently in an alpha status and the main goal is to make things work, which is the first step before making things fast.

Indeed, an article by Sébastien Deleuze that introduces the latest 0.7.0 release of the feature talks about the optimization-related work targeted for the future releases, which means that work hasn’t been done yet. So, consider the following results to be reasonably nonoptimized numbers.

Following the instructions from the repository, I built the “actuator-webmvc” sample. The build process printed some stats, which looked encouraging:

```
Build memory: 7.08GB
Image build time: 281.9s
RSS memory: 116.6M
Image size: 95.4M
Startup time: 0.211 (JVM running for 0.215)
```

That’s the difference: The build process startup time for a Spring application using `webmvc` with the actuator takes 200 ms while

consuming less than 128 MB memory. Note that this is resident set size (RSS) memory, not the heap memory.

I put the executable into a Docker image. Here's the Dockerfile, which used the `distroless` container, a small image that doesn't contain much more than the libraries applications link to. I used the multistage build to add `libz`, which is currently required by the native image.

```
FROM debian:stable-slim AS build-env
FROM gcr.io/distroless/base
COPY actuator-webmvc /app
COPY --from=build-env /lib/x86_64-linux-gnu/l
ENTRYPOINT ["/app"]
```

Running this image gave this result:

```
```
> docker run --rm --memory 128M -p 8083:8080

  .   ____   _   __ _ _
 /\\ / ___'_ __ _ _(_)_ __ __ _ \ \ \ \
(( )___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/ ___)| |_)| | | | | || (_| |))))
  ' |____| .__|_| |_|_| |___, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::

2020-06-16 23:42:11.327 INFO 1 --- [
2020-06-16 23:42:11.328 INFO 1 --- [
2020-06-16 23:42:11.763 INFO 1 --- [
Jun 16, 2020 11:42:11 PM org.apache.coyote.Ab
INFO: Initializing ProtocolHandler ["http-nio
Jun 16, 2020 11:42:11 PM org.apache.catalina.
INFO: Starting service [Tomcat]
Jun 16, 2020 11:42:11 PM org.apache.catalina.
INFO: Starting Servlet engine: [Apache Tomcat
Jun 16, 2020 11:42:11 PM org.apache.catalina.
INFO: Initializing Spring embedded WebApplica
2020-06-16 23:42:11.767 INFO 1 --- [
2020-06-16 23:42:11.771 WARN 1 --- [
2020-06-16 23:42:11.820 INFO 1 --- [
2020-06-16 23:42:11.845 INFO 1 --- [
Jun 16, 2020 11:42:11 PM org.apache.coyote.Ab
INFO: Starting ProtocolHandler ["http-nio-808
2020-06-16 23:42:11.870 INFO 1 --- [
2020-06-16 23:42:11.884 INFO 1 --- [
2020-06-16 23:42:21.923 INFO 1 --- [nio-8080
2020-06-16 23:42:21.924 INFO 1 --- [nio-8080

Started DemoApplication in 0.573 seconds (JVM
```

If the application is configured with explicit functional bean definitions instead of annotations, the startup time can be reduced even further:

```
~/repo/java-magazine-framework-comparison/spr
> docker run --rm --memory 128M -p 8083:8080
```
```

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::

Jun 17, 2020 8:49:47 AM org.springframework.b
INFO: Starting application on e4c2037ec2d8 wi
Jun 17, 2020 8:49:47 AM org.springframework.b
INFO: No active profile set, falling back to
Jun 17, 2020 8:49:47 AM org.springframework.b
INFO: Started application in 0.018 seconds (J
jafu running
```

That's fast!

Conclusion

This article explored some popular frameworks in the Java ecosystem. I ran the frameworks in a rather constrained environment fixing the available memory at 128 MB (because that can be the cheapest option in the cloud for serverless workloads), and I recorded the self-reported startup time.

Running a Spring application using GraalVM Native Image and using the experimental Spring-GraalVM-native project easily outperformed other frameworks not running on GraalVM Native Image, even though those frameworks support it. Results show that choosing a framework to get better performance for cloud workloads might mean using GraalVM Native Image.

Oleg Šelajev

Oleg Šelajev (@shelajev) is a developer advocate at Oracle Labs, working on GraalVM—the high-performance embeddable polyglot virtual machine. He organizes the VirtualJUG, the online Java User Group, and a GDG chapter in Tartu, Estonia. In his spare time, he is pursuing a PhD in dynamic system updates and code evolution. He became a Java Champion in 2017.

Share this Page