Graalkus

Let Quarkus fly high with GraalVM

Table of Contents

Introduction . . . 2
- Conversion approach . . . 2
- Demo scenario . . . 3

Part I - Development . . . 5
- Project Initialization (JIT) . . . 5
- Going AOT . . . 6
- Container images . . . 9
- Benchmark application . . . 10
- Profile-Guided Optimizations . . . 13
- Conclusion . . . 16

Part II - CI and container images . . . 18
- JIT Container image . . . 18
- AOT Container image . . . 18
- AOT Optimized Container image . . . 19
- Benchmark with limits . . . 19
- Benchmark with API gateway . . . 22

Part III - OpenShift . . . 27
- Graalkus configuration . . . 27
- Graalkus deployment . . . 27
- OpenShift benchmark . . . 27

Appendix A : Going AOT in depth . . . 30

Appendix B : Resources . . . 32

Appendix C : Quarkus OpenShift Extension . . . 33
- Installation . . . 33
- Configuration . . . 33
- Deployment . . . 33

This project has been build with a doc-as-code approach from the repository : https://github.com/fugerit-org/graalkus.

The cover image of the PDF version has been generated with the help of DALL·E

Introduction

In recent years the interest for going AOT has grown more and more among the java developers community.

Some projects were born or added support for GraalVM and native compilation. Just to name a few :

- Quarkus
- Spring Boot
- Micronaut
- Helidon

Using GraalVM has some great benefits (for instance faster startup and lower memory footprint) and a few limitations (configuration complexity, runs only on target environment).

AOT may not be viable for all scenarios, but when it is possible to use if performances and costs can be reduced a lot.

Starting in 2023 I've been using it more and more on the projects I'm working on.

Talking with other developers interested in the technology, one big obstacle to GraalVM adoption is first of all configuration complexity (for features like reflection).

Graalkus is a simple microservice, based on Quarkus, that I created to share my personal experience on migrating JIT application to AOT.

Conversion approach

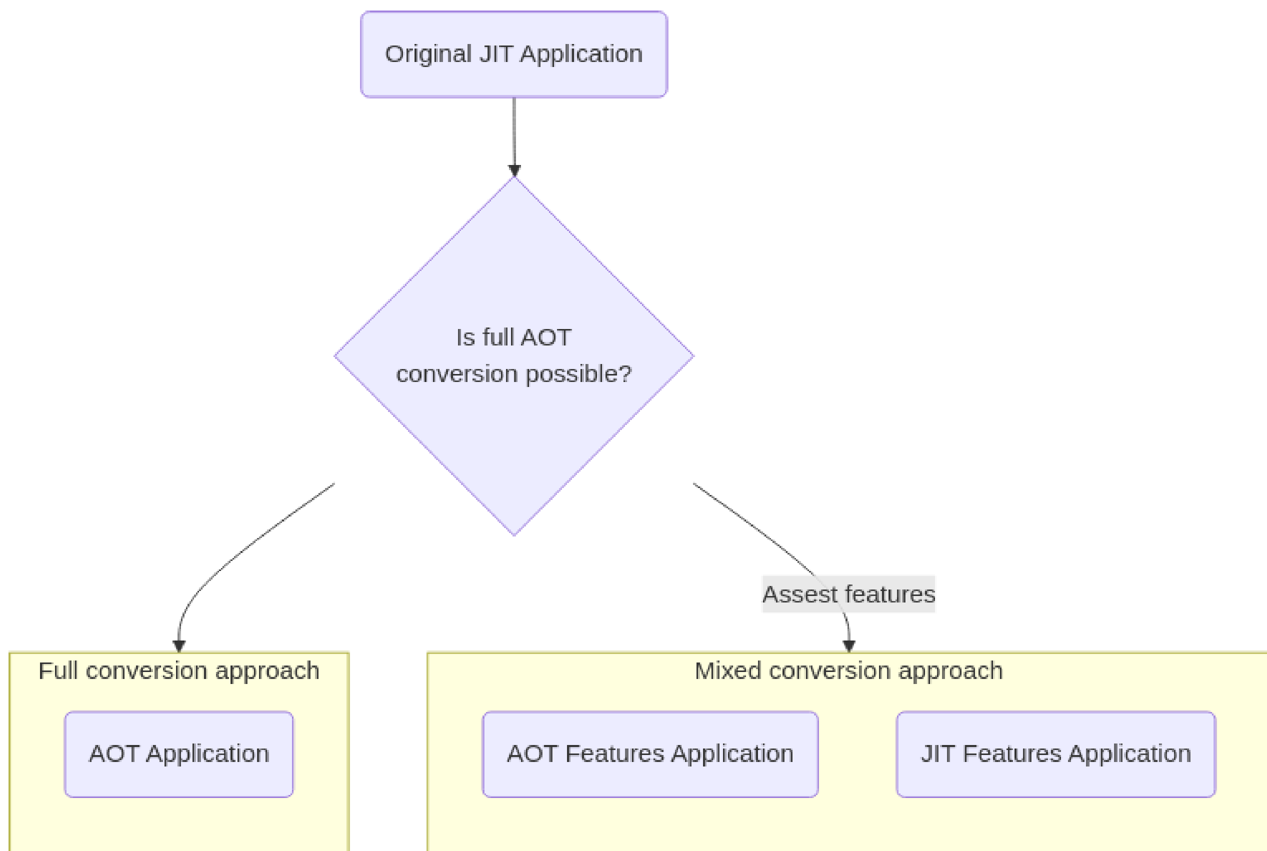

Figure 1. JIT to AOT conversion approach

Usually I took into consideration two possible approach when migrating a JIT application to AOT :

1. Full approach - when all the features can be easily configured to be included in a native build
2. Mixed approach - when not all features can be converted to AOT for any reason, for instance :
   - costs - we need to rewrite the feature and we decide conversion is not worth
   - technical limitation - some feature simply relies on some technology which cannot be converted (i.e. a very old library)

Often the mixed approach could be a good idea, because conversion can be sometimes complex and it is easier to isolate the features to be converted. Maybe starting from the easier and iterate on the others in a later time.

Demo scenario

This demo is inspired by a real microservice I migrated to AOT some time ago.

The scenario we take in consideration is a JIT application used generate documents in various formats (HTML, MarkDown, AsciiDoc and PDF), through rest services.

Let’s define every format as a feature, and the load is roughly this :

1. HTML : 40%

2. MarkDown : 30%
3. AsciiDoc : 20%
4. PDF : 10%

We will find out that PDF conversion it is not easy to implement.

So we will use the mixed approach, converting formats 1, 2, 3 only. So at the end the AOT Application will handle the 90% of the load, whereas the JIT Application will be left with only 10%.

We can use an API gateway or some other technology to keep usage transparent for clients.

Part I - Development

In this section we will describe how to develop our demo application :

1. Project initialization (JIT)

Requirements

- Oracle GraalVM (tested on 21.0.5)
- Apache Maven (tested on 3.9.9)
- Container environment (i.e. docker, podman)

Project Initialization (JIT)

We will create a project based on Venus, a Framework to produce documents in different output formats starting from an XML document model.

Venus has a maven plugin to initialize a maven project with some flavours. I'm going to pick a Quarkus application with the command :

```
mvn org.fugerit.java:fj-doc-maven-plugin:init \
-DgroupId=org.fugerit.java.demo \
-DartifactId=graalkus \
-Dflavour=quarkus-3 \
-Dextensions=base,freemarker,mod-fop
```

This will create a maven project structure, with a rest service for document generation in html, adoc, markdown and pdf format.

Just run :

```
mvn quarkus:dev
```

And access the swagger ui to check available paths :

http://localhost:8080/q/swagger-ui/

For instance the PDF version http://localhost:8080/doc/example.pdf or the AsciiDoc one http://localhost:8080/doc/example.adoc.

Ready for the next step?

Going AOT

As stated in Quarkus documentation, we try to build a native executable running :

```
mvn install -Dnative
```

Which will lead to a few errors, starting with :

```
Error: Detected a started Thread in the image heap. Thread name: Java2D Disposer.
Threads running in the image generator are no longer running at image runtime. If
these objects should not be stored in the image heap, you can use

    '--trace-object-instantiation=java.lang.Thread'
```

It is often possible to achieve AOT compatibility tweaking a few parameters. GraalVM is very good at providing hints on what to do (like in the example above). There are also a few techniques helping to configure the application in order to create a native image (for instance the tracing agent).

Generally speaking the framework we are using, Venus, is already configured for AOT. Unfortunately not all modules are native ready. In particular the mod-fop extension it is not easy to be built with GraalVM.

This is partly explained in a Quarkus Camel issue about pdfbox 2.

Our goal is to show a demo for the mixed JIT to AOT conversion approach.

In this scenario we modify the applicatin to run both in JIT and AOT mode, but in latter the PDF document feature will be disabled.

We will achieve with three simple modifications.

1. Update the maven pom file

The main reason why we get the error is that GraalVM fails on this dependency at build time :

```
    <dependency>
      <groupId>org.fugerit.java</groupId>
      <artifactId>fj-doc-mod-fop</artifactId>
      <exclusions>
        <exclusion>
          <groupId>xml-apis</groupId>
          <artifactId>xml-apis</artifactId>
        </exclusion>
      </exclusions>
```

```
    </dependency>
```

So we will move it to the profiles sectionas and make it only available in JIT profile :

```
<profile>
      <id>jit</id>
      <activation>
        <property>
          <name>!native</name>
        </property>
      </activation>
      <dependencies>
        <dependency>
          <groupId>org.fugerit.java</groupId>
          <artifactId>fj-doc-mod-fop</artifactId>
          <exclusions>
            <exclusion>
              <groupId>xml-apis</groupId>
              <artifactId>xml-apis</artifactId>
            </exclusion>
          </exclusions>
        </dependency>
      </dependencies>
    </profile>
    <profile>
      <id>native</id>
      <activation>
        <property>
          <name>native</name>
        </property>
      </activation>
      <properties>
        <skipITs>true</skipITs>
        <quarkus.native.enabled>true</quarkus.native.enabled>
      </properties>
      <dependencies>
        <dependency>
          <groupId>org.fugerit.java</groupId>
          <artifactId>fj-doc-mod-fop</artifactId>
          <scope>provided</scope>
          <exclusions>
            <exclusion>
              <groupId>xml-apis</groupId>
              <artifactId>xml-apis</artifactId>
            </exclusion>
          </exclusions>
        </dependency>
      </dependencies>
```

```
    </profile>
```

Of course the PDF document using Apache FOP will fail when running the native executables.

2. Update the project config file

This application use the file **src/main/resources/graalkus/fm-doc-process-config.xml** to load doc handlers classes by reflection, we will mark as **unsafe** the handlers not available in AOT mode :

```
<freemarker-doc-process-config>
<!-- Type handler generating xls:fo style sheet -->
<docHandler id="fo-fop" info="fo"
type="org.fugerit.java.doc.mod.fop.FreeMarkerFopTypeHandlerUTF8" unsafe="true"/>
<!-- Type handler generating pdf -->
<docHandler id="pdf-fop" info="pdf"
type="org.fugerit.java.doc.mod.fop.PdfFopTypeHandler" unsafe="true">
</freemarker-doc-process-config>
```

3. Disable relevant tests

We are going to add the JUnit 5 DisabledInNativeImage annotation to the tests that would fail :

```
    @Test
    @DisabledInNativeImage
    void testPdf() {
        given().when().get("/doc/example.pdf").then().statusCode(200);
    }
```

So now we can try again to build native image :

```
mvn package -Dnative
```

This time the build is successful and features for HTML, AsciiDoc and MarkDown documents will be available, for instance http://localhost:8080/doc/example.adoc, while the pdf version will fail http://localhost:8080/doc/example.pdf.

So we have now a project which can be built both in JIT and AOT mode.

Now it’s time for the docker images.

Container images

In this step we are going to build and test the container image.

JIT container

First of all we build the application :

```
mvn package
```

Then build the container image :

```
docker build -f src/main/docker/Dockerfile.jvm -t graalkus-jit .
```

And launch it :

```
docker run --rm -p 8080:8080 --name graalkus-jit graalkus-jit
```

On my system quarkus starts in 0.458s.

```
__  ____  __  _____   ___  __ ____  ______
--/ __ \/ / / / _ | / _ \/ //_/ / / / __/
-/ /_/ / /_/ / __ |/ , _/ ,< / /_/ /\ \
--\___\_\____/_/ |_/_/|_/_/|_|\____/___/
2024-12-01 00:50:30,285 INFO  [org.fug.jav.dem.gra.AppInit] (main) The application
is starting...
2024-12-01 00:50:30,333 INFO  [io.quarkus] (main) graalkus 1.0.0-SNAPSHOT on JVM
(powered by Quarkus 3.17.2) started in 0.458s. Listening on: http://0.0.0.0:8080
```

AOT container

After building the application :

```
mvn package -Dnative -Dquarkus.native.container-build=true
```

this time we are going to use the quarkus.native.container-build option, so the build will be handled by a container.

We can now build the container :

```
docker build -f src/main/docker/Dockerfile.native-micro -t graalkus-aot .
```

And launch it :

```
docker run --rm -p 8080:8080 --name graalkus-aot graalkus-aot
```

This time quarkus starts in 0.020s, about 25 times faster than JIT version!

```
__  ____  __  _____   ___  __ ____  ______
 --/ __ \/ / / / _ | / _ \/ //_/ / / / __/
 -/ /_/ / /_/ / __ |/ , _/ ,< / /_/ /\ \
--\___\_\____/_/ |_/_/|_/_/|_|\____/___/
2024-12-01 00:52:13,027 INFO  [org.fug.jav.dem.gra.AppInit] (main) The application
is starting...
2024-12-01 00:52:13,029 INFO  [io.quarkus] (main) graalkus 1.0.0-SNAPSHOT native
(powered by Quarkus 3.17.2) started in 0.020s. Listening on: http://0.0.0.0:8080
```

Benchmark application

In this step we are going to benchmark the application, both in JIT and AOT version.

Requirements

For this benchmark we will use a script that can be found in the folder bench-graph-h2-load.sh, it is possible to find it in the following path of the repository *src/main/script/bench-graph-h2-load.sh*.

The script needs :

- siege, for actual benchmark
- psrecord, to plot resources usage

Benchmark JIT application

Build the application

```
mvn package
```

Run the script (will also launch the application)

```
./src/main/script/bench-graph-siege.sh -m JIT
```

Benchmark AOT application

Build the application

```
mvn install -Dnative
```

Run the script (will also launch the application)

```
./src/main/script/bench-graph-siege.sh -m AOT
```

Sample output

Here I will show, as an example, the result on my system.

- OS : Ubuntu 24.04.1 LTS
- CPU : AMD Ryzen 7 3700X (8 core, 16 thread)
- Memory : 32 GB

With standard script parameters (h2load) :

- 50000 requests for warm up run (w)
- 250000 requests for benchmark run (r)
- 12 clients (c)
- 1 threads (t)

JIT result :

```
{
    "transactions":                 333324,
    "availability":                 100.00,
    "elapsed_time":                  25.92,
    "data_transferred":             404.11,
    "response_time":                  0.00,
    "transaction_rate":           12859.72,
    "throughput":                    15.59,
    "concurrency":                   11.31,
    "successful_transactions":          333324,
    "failed_transactions":                   0,
    "longest_transaction":                0.07,
    "shortest_transaction":               0.00
}
```

AOT result :

```
{
    "transactions":                 333324,
    "availability":                 100.00,
    "elapsed_time":                  34.78,
    "data_transferred":             404.11,
    "response_time":                  0.00,
    "transaction_rate":            9583.78,
    "throughput":                    11.62,
    "concurrency":                   11.43,
    "successful_transactions":          333324,
    "failed_transactions":                   0,
    "longest_transaction":                0.13,
    "shortest_transaction":               0.00
}
```


And the relative resource plotting :

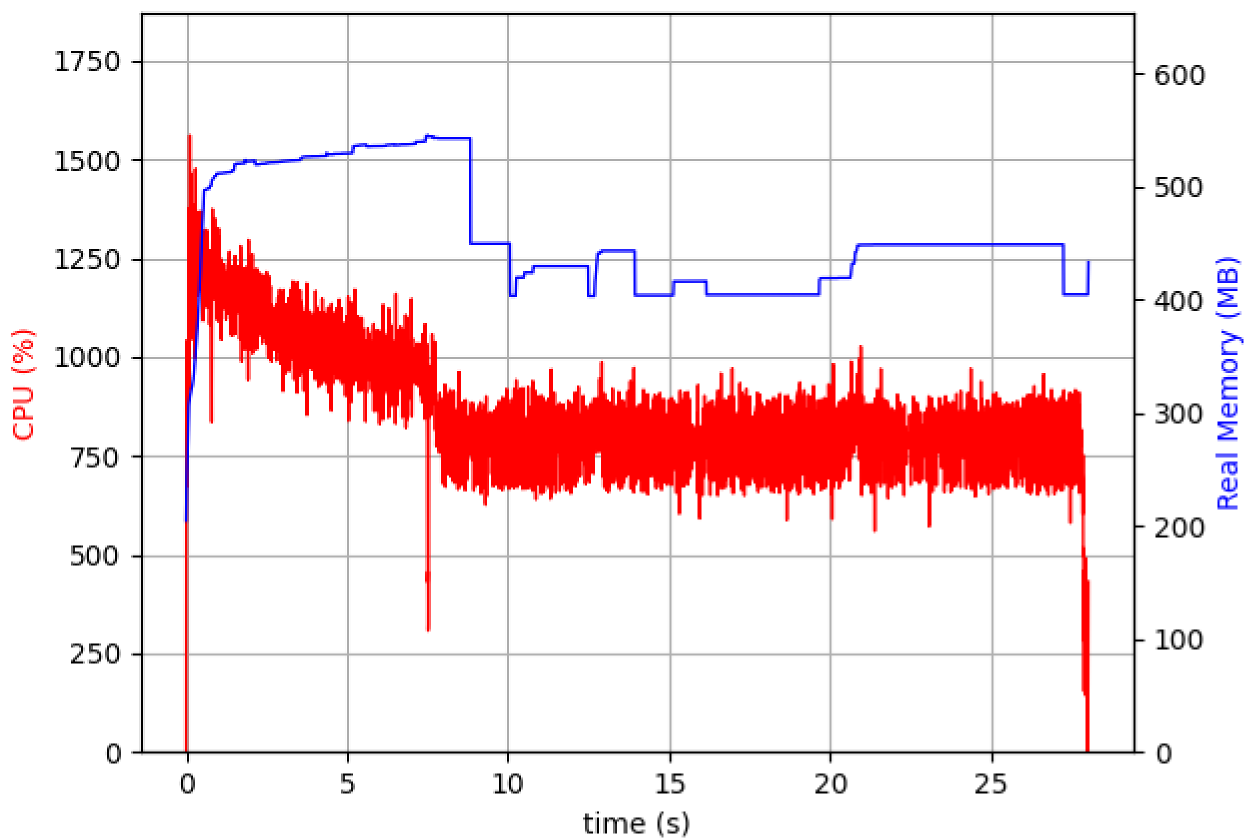

Figure 2. JIT Benchmark plotting

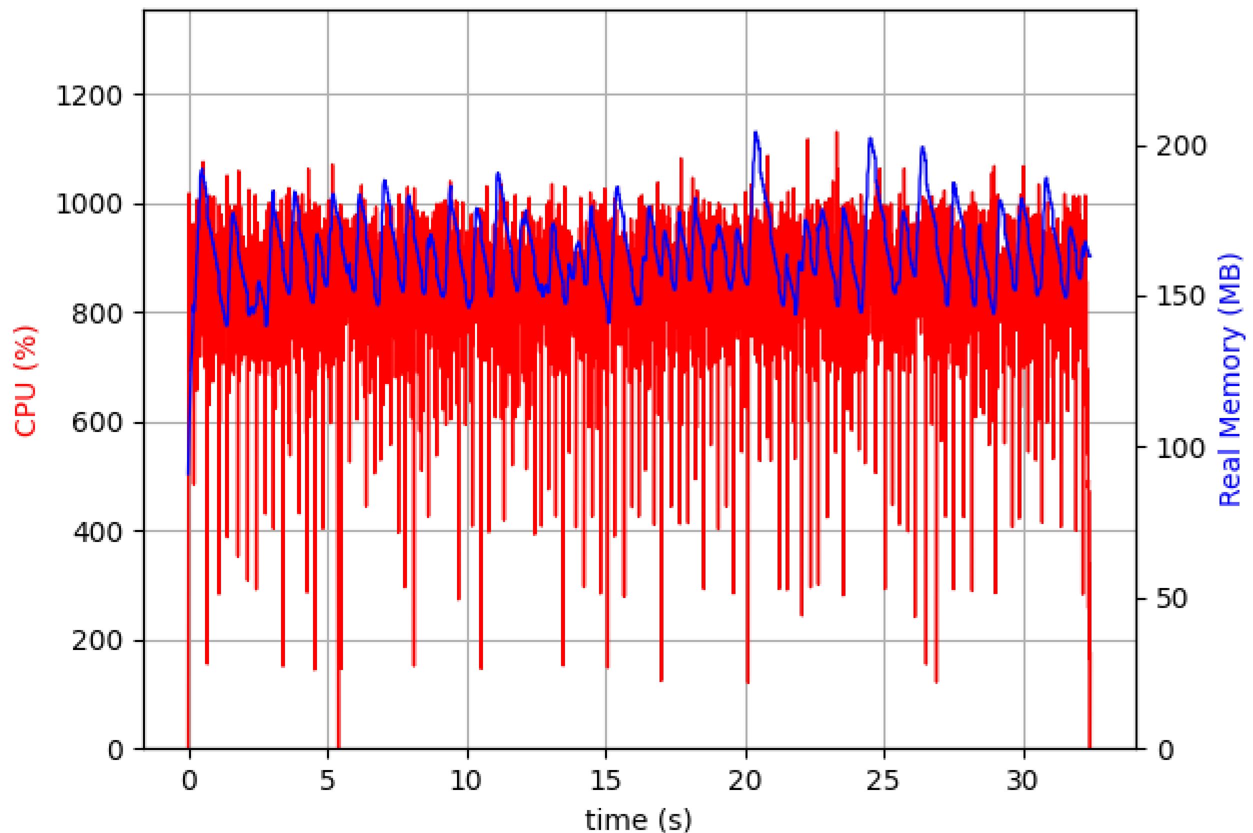

Figure 3. AOT Benchmark plotting

As you can see :

- The rate is more or less the same for JIT and AOT version
- All request are successful in both scenarios
- CPU footprint is also comparable (Except at startup where AOT performs better)
- AOT memory footprint is 3x times lower than JIT version

Keep in mind we did not add any optimization to JIT version (for instance CRaC), nor to AOT one (i.e. PGO).

Profile-Guided Optimizations

Native executables with GraalVM can perform better if they are optimized with some real data.

In this section we will explore the GraaVM's Profile-Guided Optimizations feature.

Instrumentation

1. We add an instrumented profile to our project :

```
<profile>
  <id>instrumented</id>
  <build>
    <finalName>${project.artifactId}-${project.version}-instrumented</finalName>
  </build>
  <properties>
    <quarkus.native.additional-build-args>${base-native-build-args},--pgo-
instrument</quarkus.native.additional-build-args>
  </properties>
</profile>
```

1. Then we will create the native image :

```
mvn install -Dnative -Pinstrumented
```

1. Start the application :

```
./target/graalkus-*-instrumented-runner
```

1. Provide some relevant workload :

```
./src/main/script/bench-graph-siege.sh
```

After the application shutdown a *.iprof* file will be available in the working folder.

Optimization

1. Add another profile to build the optimized native image :

```
<profile>
  <id>optimized</id>
  <build>
    <finalName>${project.artifactId}-${project.version}-optimized</finalName>
  </build>
  <properties>
    <quarkus.native.additional-build-args>${base-native-build-args},--
pgo=${project.basedir}/default.iprof</quarkus.native.additional-build-args>
  </properties>
</profile>
```

1. Create the optimized native executable :

```
mvn install -Dnative -Poptimized
```

1. Run the benchmark :

```
./src/main/script/bench-graph-siege.sh -m AOT -a graalkus-*-optimized-runner
```

1. Sample optimized result

This section contains the result of an optimized benchmark run :


```
{
    "transactions":                 333324,
    "availability":                 100.00,
    "elapsed_time":                  25.92,
    "data_transferred":             404.11,
    "response_time":                  0.00,
    "transaction_rate":           12859.72,
    "throughput":                    15.59,
    "concurrency":                   11.33,
    "successful_transactions":          333324,
    "failed_transactions":                   0,
    "longest_transaction":                0.05,
    "shortest_transaction":               0.00
}
```


And the relative resource plotting :

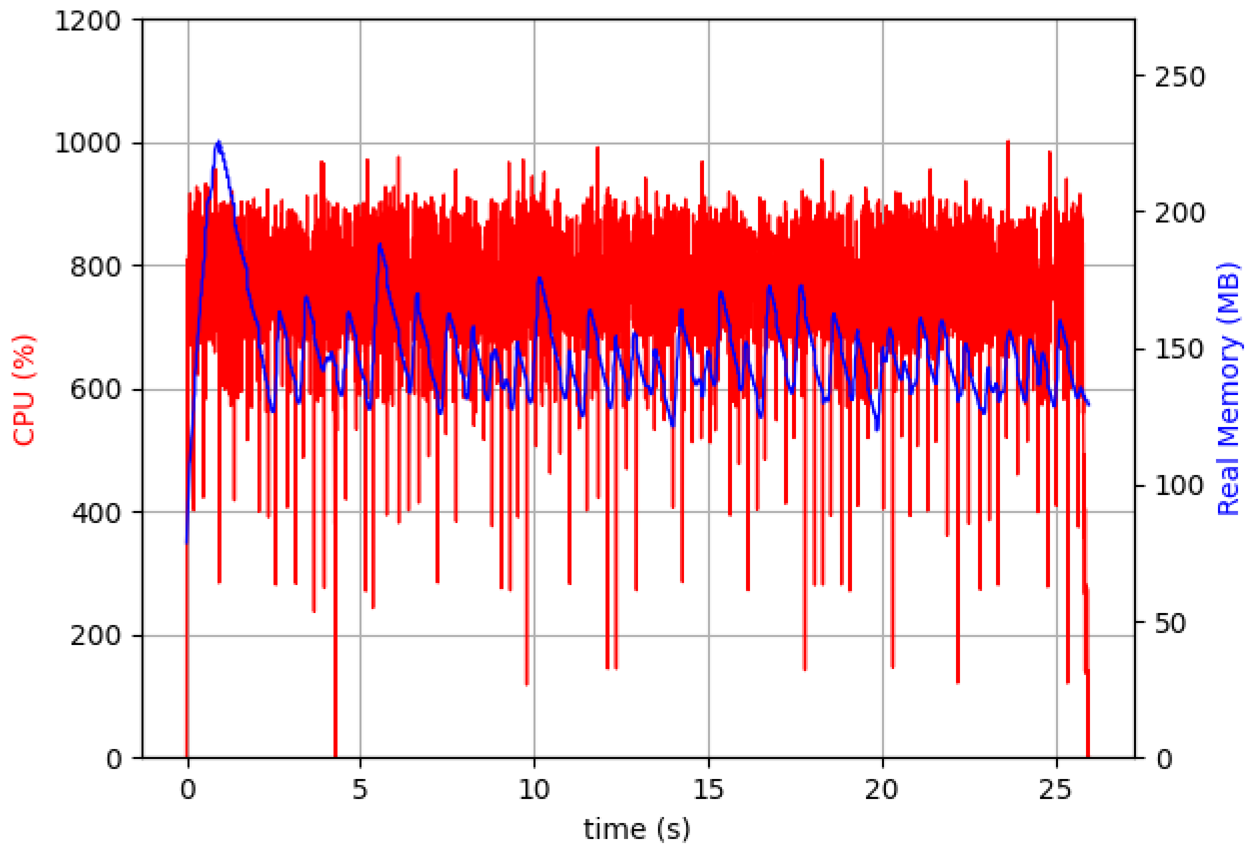

Figure 4. Optimized AOT Benchmark plotting

Let's compare the result with the Unoptimized benchmark (they have been run on the same system).

After optimization, CPU and memory footprint is more or less the same, but request rate is about 20% higher (160.94 req/s from to 197.93 req/s).

More optimization options are available. A good resource for it is the Build and test various capabilities of Spring Boot & GraalVM repository. (Even though focused on Spring Boot, most concept and options can be used on other frameworks too).

Profile-Guided Optimizations are only available on Oracle GraalVM. other distributions like GraalVM Community Edition or Mandrel do not provide it.

Conclusion

So in this first part we :

1. Developed the stand alone JIT application
2. Converted it to an AOT application

3. Created the container image version of each
4. Run benchmarks on standalone application
5. Done native image optimization (PGO)

Here is a summary of the result :

Info	JIT	AOT	Optimized AOT
Startup time (s)	0.634	0.018	0.014
Requests/s	19068.33	14819.13	19464.30
Memory (MB)	400/500	150/250	150/250

Part II - CI and container images

This section describes container images build thought CI :

- JIT Version
- AOT Version
- AOT PGO Version

JIT Container image


```
{
  "transactions": 333324,
  "availability": 100,
  "elapsed_time": 31.52,
  "data_transferred": 404.11,
  "response_time": 0,
  "transaction_rate": 10575,
  "throughput": 12.82,
  "concurrency": 11.41,
  "successful_transactions": 333324,
  "failed_transactions": 0,
  "longest_transaction": 0.01,
  "shortest_transaction": 0
}
```


AOT Container image


```
{
  "transactions": 333324,
  "availability": 100,
  "elapsed_time": 55.27,
  "data_transferred": 404.11,
  "response_time": 0,
  "transaction_rate": 6030.83,
  "throughput": 7.31,
  "concurrency": 11.69,
  "successful_transactions": 333324,
  "failed_transactions": 0,
  "longest_transaction": 0.02,
  "shortest_transaction": 0
}
```

AOT Optimized Container image


```
{
  "transactions": 333324,
  "availability": 100,
  "elapsed_time": 32.79,
  "data_transferred": 404.11,
  "response_time": 0,
  "transaction_rate": 10165.42,
  "throughput": 12.32,
  "concurrency": 11.44,
  "successful_transactions": 333324,
  "failed_transactions": 0,
  "longest_transaction": 0.02,
  "shortest_transaction": 0
}
```


Benchmark with limits

Let’s configure a docker compose to limit resources for our containers :

```
# docker compose -f src/main/docker/docker-compose-limit.yml up -d

# Define the services
services:
  graalkus-jit-limit:
    # it is possible to pick more options from :
    # https://hub.docker.com/repository/docker/fugeritorg/graalkus/general
    # or build your own image locally.
    image: fugeritorg/graalkus:latest
    container_name: graalkus-jit-limit
    restart: always
    ports:
      - "9084:8080"
    deploy:
      resources:
        limits:
          cpus: 1.0
          memory: 128M
        reservations:
          cpus: 1.0
          memory: 64M
  graalkus-aot-limit:
    # it is possible to pick more options from :
    # https://hub.docker.com/repository/docker/fugeritorg/graalkus/general
    # or build your own image locally.
```

```
    image: fugeritorg/graalkus:latest-native
    container_name: graalkus-aot-limit
    restart: always
    ports:
      - "9085:8080"
    deploy:
      resources:
        limits:
          cpus: 1.0
          memory: 128M
        reservations:
          cpus: 1.0
          memory: 64M
  graalkus-aot-optimized-limit:
    # it is possible to pick more options from :
    # https://hub.docker.com/repository/docker/fugeritorg/graalkus/general
    # or build your own image locally.
    image: fugeritorg/graalkus:latest-native-pgo
    container_name: graalkus-aot-optimized-limit
    restart: always
    ports:
      - "9086:8080"
    deploy:
      resources:
        limits:
          cpus: 1.0
          memory: 128M
        reservations:
          cpus: 1.0
          memory: 64M
  graalkus-jit-limit-high:
    image: fugeritorg/graalkus:latest
    container_name: graalkus-jit-high-limit
    restart: always
    ports:
      - "9087:8080"
    deploy:
      resources:
        limits:
          cpus: 1.0
          memory: 256M
        reservations:
          cpus: 1.0
          memory: 64M
```

and start it the containers :

```
docker compose -f src/main/docker/docker-compose-limit.yml up -d
```

For this compose configuration the pre-built container images.

Then benchmark one by one the services :

1. JIT Version (1.0 CPU, max 32/128 MB)

```
./src/main/script/bench-graph-siege.sh -u http://localhost:9084
```


```
{
  "transactions": 333324,
  "availability": 100,
  "elapsed_time": 158.11,
  "data_transferred": 404.11,
  "response_time": 0.01,
  "transaction_rate": 2108.18,
  "throughput": 2.56,
  "concurrency": 11.87,
  "successful_transactions": 333324,
  "failed_transactions": 0,
  "longest_transaction": 0.08,
  "shortest_transaction": 0
}
```


2. AOT Version (1.0 CPU, max 32/64 MB)

```
./src/main/script/bench-graph-siege.sh -u http://localhost:9085
```


```
{
  "transactions": 333324,
  "availability": 100,
  "elapsed_time": 352.7,
  "data_transferred": 404.11,
  "response_time": 0.01,
  "transaction_rate": 945.06,
  "throughput": 1.15,
  "concurrency": 11.92,
  "successful_transactions": 333324,
  "failed_transactions": 0,
  "longest_transaction": 0.1,
  "shortest_transaction": 0
```

```
}
```

3. AOT Optimized Version (1.0 CPU, max 32/64 MB)

```
./src/main/script/bench-graph-siege.sh -u http://localhost:9086
```


```
{
  "transactions": 333324,
  "availability": 100,
  "elapsed_time": 187.58,
  "data_transferred": 404.11,
  "response_time": 0.01,
  "transaction_rate": 1776.97,
  "throughput": 2.15,
  "concurrency": 11.89,
  "successful_transactions": 333324,
  "failed_transactions": 0,
  "longest_transaction": 0.09,
  "shortest_transaction": 0
}
```


Benchmark with API gateway

In the end here is a configuration with an API gateway. We are going to use Traefik.

```
# docker compose -f src/main/docker/docker-compose-mixed.yml up -d

networks:
  graalkus_network:
    driver: bridge
    ipam:
      config:
        - subnet: 172.80.0.0/16

# Define the services
services:

  api-gateway-mixed:
    image: traefik:v2.5
    container_name: graalkus-api-mixed
    volumes:
      - ${PWD}/src/main/docker/traefik-mixed/traefik.yml:/etc/traefik/traefik.yml
      - /var/run/docker.sock:/var/run/docker.sock
    networks:
```

```
      - graalkus_network
    ports:
      - "8088:80"
      - "8089:8080"

  graalkus-jit-mixed:
    # it is possible to pick more options from :
    # https://hub.docker.com/repository/docker/fugeritorg/graalkus/general
    # or build your own image locally.
    image: fugeritorg/graalkus:v1.2.1
    container_name: graalkus-jit-mixed
    hostname: jit
    labels:
      - traefik.http.routers.jit.rule=Path(`/doc/example.pdf`)
    restart: always
    networks:
      - graalkus_network
    ports:
      - "8086:8080"
    deploy:
      resources:
        limits:
          cpus: 1.0
          memory: 256M
        reservations:
          cpus: 1.0
          memory: 64M

  graalkus-aot-mixed:
    # it is possible to pick more options from :
    # https://hub.docker.com/repository/docker/fugeritorg/graalkus/general
    # or build your own image locally.
    image: fugeritorg/graalkus:v1.2.1-native-pgo
    container_name: graalkus-aot-mixed
    hostname: aot
    labels:
      - traefik.http.routers.aot.rule=PathPrefix(`/doc`)
    restart: always
    networks:
      - graalkus_network
    ports:
      - "8087:8080"
    deploy:
      resources:
        limits:
          cpus: 1.0
          memory: 64M
        reservations:
          cpus: 1.0
```

```yaml
          memory: 32M

  graalkus-jit-std:
    # it is possible to pick more options from :
    # https://hub.docker.com/repository/docker/fugeritorg/graalkus/general
    # or build your own image locally.
    image: fugeritorg/graalkus:v1.2.1
    container_name: graalkus-jit-mixed-std
    hostname: jit-std
    restart: always
    networks:
      - graalkus_network
    ports:
      - "8085:8080"
    deploy:
      resources:
        limits:
          cpus: 2.0
          memory: 512M
        reservations:
          cpus: 1.0
          memory: 128M
```

and start the containers :

```
docker compose -f src/main/docker/docker-compose-mixed.yml up -d
```

For this compose configuration the pre-built container images.

all the test up now were run only on functions both supported by JIT and AOT version of the application. The benchmark in this section will be run with the all the features enabled, using the *-p* flag.

Then benchmark one by one the services :

1. API Gateway version

Resources total :

- Min CPU : 1.0
- Min memory : 96M
- Max CPU : 2.0
- Max memory : 320M

In this scenario :

- Traefik will be used as api gateway.
- /doc/example.pdf path will be served by JIT application.
- All other urls will be served by AOT application.

```
./src/main/script/bench-graph-siege.sh -p -u http://localhost:8088
```


```
{
  "transactions": 300000,
  "availability": 100,
  "elapsed_time": 155.46,
  "data_transferred": 613.37,
  "response_time": 0.01,
  "transaction_rate": 1929.76,
  "throughput": 3.95,
  "concurrency": 11.86,
  "successful_transactions": 300000,
  "failed_transactions": 0,
  "longest_transaction": 1.14,
  "shortest_transaction": 0
}
```


2. Pure JIT Version

In this scenario all urls are served by the JIT application.

Resources :

- Min CPU : 1.0
- Min memory : 128M
- Max CPU : 2.0
- Max memory : 512M

```
./src/main/script/bench-graph-siege.sh -p -u http://localhost:8085
```


```
{
  "transactions": 300000,
  "availability": 100,
  "elapsed_time": 145.36,
  "data_transferred": 613.37,
  "response_time": 0.01,
  "transaction_rate": 2063.84,
  "throughput": 4.22,
```

```
  "concurrency": 11.85,
  "successful_transactions": 300000,
  "failed_transactions": 0,
  "longest_transaction": 0.33,
  "shortest_transaction": 0
}
```


Conclusion

The mixed API Gateway + JIT + AOT PGO version has a 40% memory saving compared to the pure JIT version.

Part III - OpenShift

This section will configure and deploy Graalkus on a real platform. The platform of choice is RedHat Developer Sandbox, which allows to use an Openshift environment free of costs for one month.

Graalkus configuration

We need to install the SmallRye Health extension :

```
mvn quarkus:add-extension -Dextensions='smallrye-health'
```

this extension is needed to provide health check.

Graalkus deployment

From here on the OpenShift CLI is required.

After *oc* installation, from the OpenShift Sandbox profile select "Copy Login Command" and login to openshift :

```
oc login --token=sha256****************** --server=https://api.sandbox-
m4.g2pi.p1.openshiftapps.com:6443
```

At this point it is possible to create the OpenShift resources, for instance using the script :

```
src/main/openshift/oc-apply-all.sh $namespace $cluster_domain
```

OpenShift benchmark

1. Pure JIT Version

```
{
  "transactions": 300000,
  "availability": 100,
  "elapsed_time": 213.9,
  "data_transferred": 613.37,
  "response_time": 0.01,
  "transaction_rate": 1402.52,
  "throughput": 2.87,
```

```
  "concurrency": 11.42,
  "successful_transactions": 300000,
  "failed_transactions": 0,
  "longest_transaction": 1.06,
  "shortest_transaction": 0
}
```

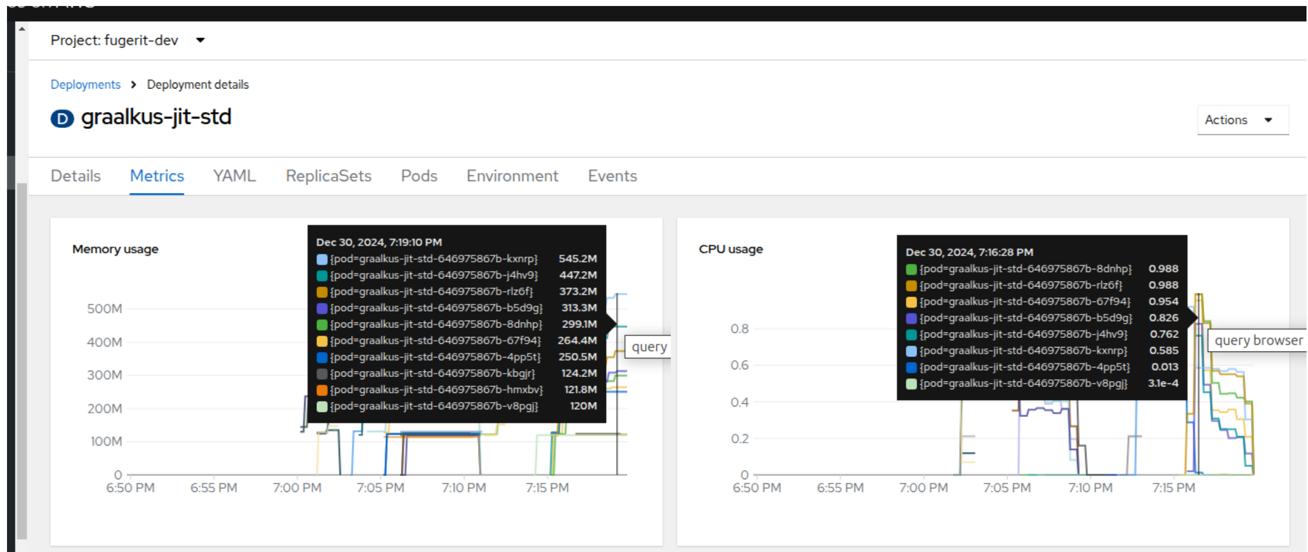

Figure 5. Pure JIT Benchmark OpenShift

2. Mixed AOT/JIT Version

```
{
  "transactions": 300000,
  "availability": 100,
  "elapsed_time": 200.06,
  "data_transferred": 613.37,
  "response_time": 0.01,
  "transaction_rate": 1499.55,
  "throughput": 3.07,
  "concurrency": 11.61,
  "successful_transactions": 300000,
  "failed_transactions": 0,
  "longest_transaction": 1.06,
  "shortest_transaction": 0
}
```

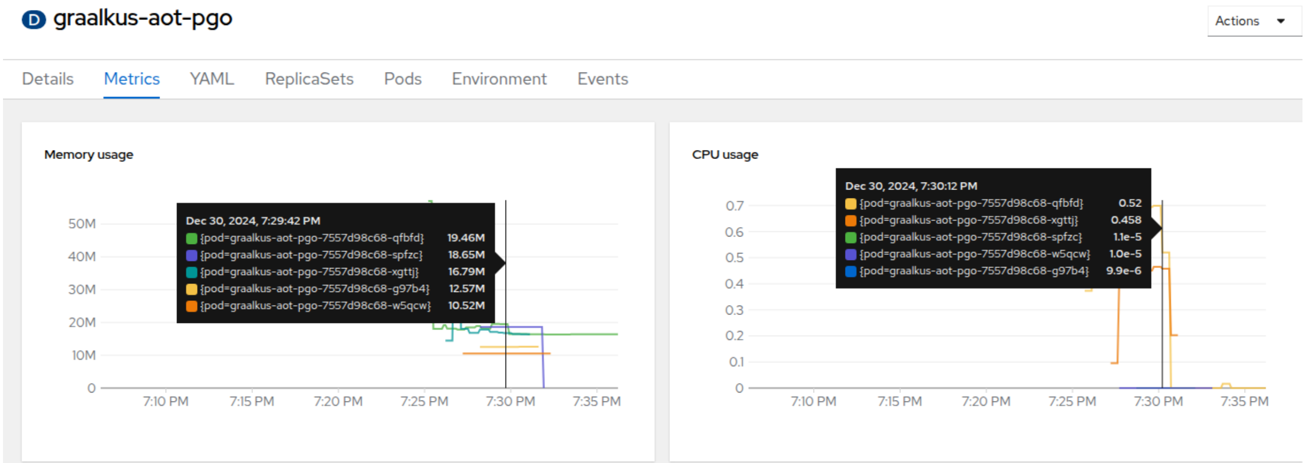

Figure 6. API AOT Benchmark OpenShift

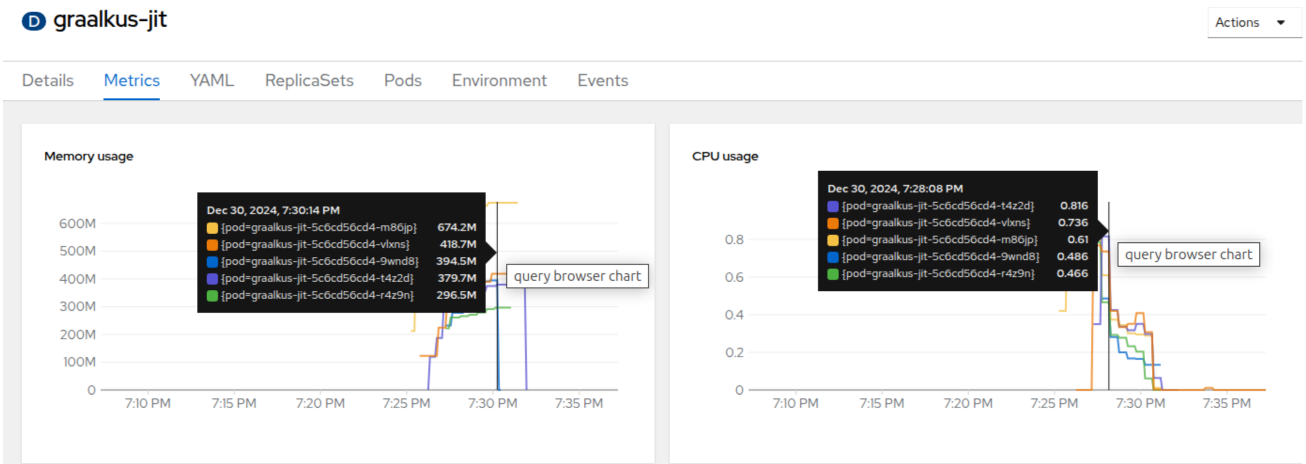

Figure 7. API JIT Benchmark OpenShift

Appendix A : Going AOT in depth

In Going AOT section we just adapted our software with a few modifications.

This was possible because :

1. The application is built on Quarkus which is already native ready. All core modules are already supporting native image build.
2. Venus framework too is already pre-configure for native image build. For instance by configuring the native build args.

```
quarkus:
  native:
    # if needed add -H:+UnlockExperimentalVMOptions
    additional-build-args: -H:IncludeResources=graalkus/fm-doc-process-config.xml,\
      -H:IncludeResources=graalkus/template/document.ftl
```

In a legacy application, not based on a native ready framework like Quarkus or Spring Boot, the conversion could be lengthier.

One possible approach could be to split a monolith features in microservices and going AOT when possible.

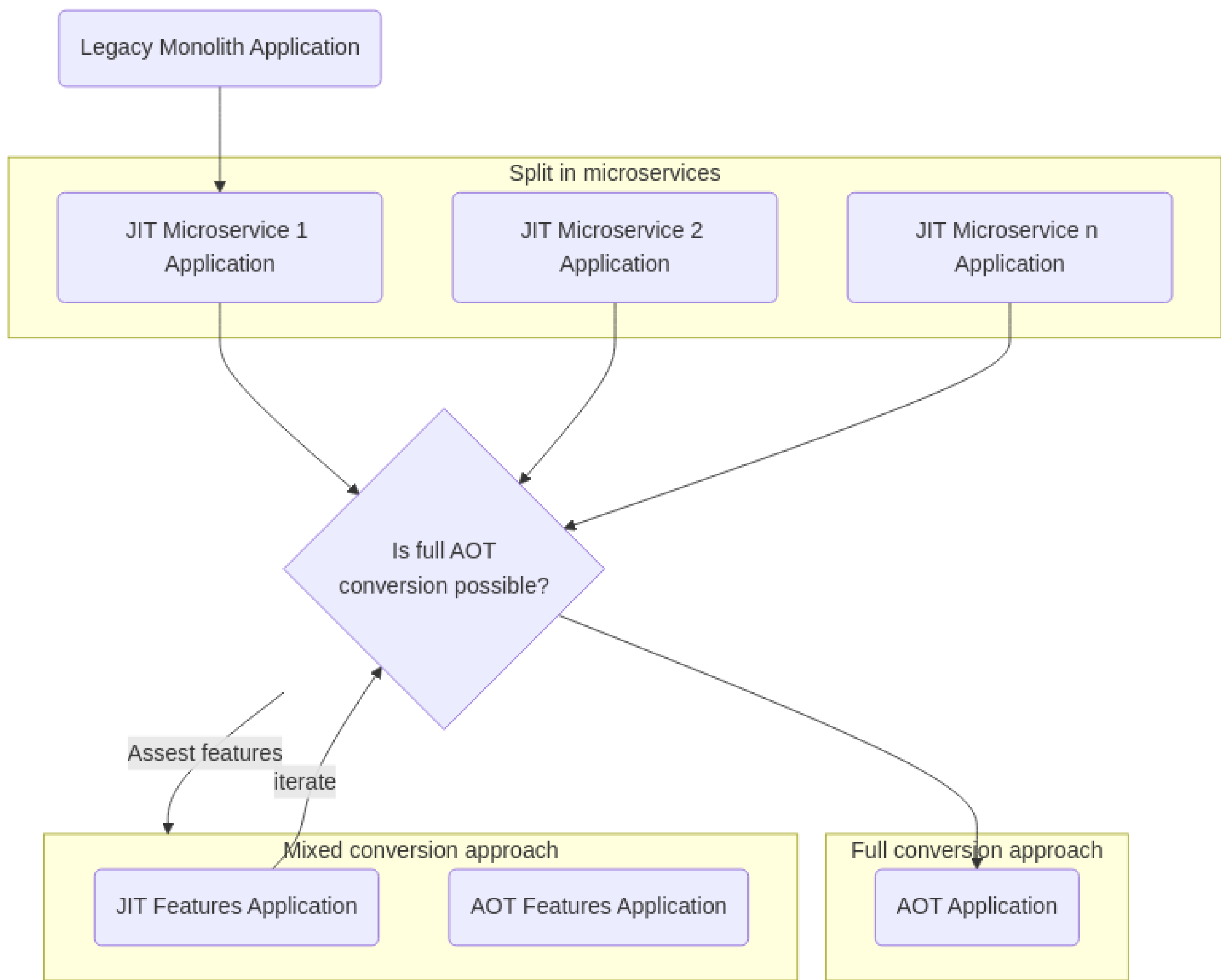

Figure 8. From legacy to AOT application

Appendix B : Resources

In this appendix there are some references to some useful documentation and resources.

1. Quarkus
   - Quarkus documentation, especially ;
     - Tips for writing native applications
     - Building a Native Executable
   - Quarkus Event Bus Logging Filter JAX-RS Documentation (a very good example of both Quarkus Event Bus usage and doc-as-code approach).
2. Fugerit Venus Doc
   - Venus Documentation
   - Venus Quarkus Tutorial App
   - Venus Online Playground
3. GraalVM
   - GraalVM documentation
     - Profile-Guided Optimizations
   - Build and test various capabilities of Spring Boot & GraalVM (GitHub repository)
   - A few videos :
     - Going AOT: Everything you need to know about GraalVM for Java applications by Alina Yurenko SpringIO
     - Bring the action: using GraalVM in production by Alina Yurenko Going AOT: Everything you need to know about GraalVM for Java applications by Alina Yurenko SpringIO
     - Scala fino a zero con Spring + GraalVM o WebAssembly di Sébastien Deleuze

Appendix C : Quarkus OpenShift Extension

Section Graalkus deployment covered installation of Graalkus project using Kubernetes descriptors.

Alternatively it is possible to use the built-in Quarkus OpenShift extension

Installation

Let's install the

```
mvn quarkus:add-extension -Dextensions='quarkus-openshift'
```

Configuration

And configure the route exposure :

```
quarkus:
  openshift:
    route:
      expose: true
```

Deployment

First oc login is needed :

```
oc login --token=$token --server=$server
```

Then we install the JIT version

```
quarkus build \
-Dquarkus.openshift.deploy=true \
-Dquarkus.container-image.name=graalkus-ocp-jit \
-Dquarkus.openshift.name=graalkus-ocp-jit
```

And the AOT version

```
quarkus build --native \
-Dquarkus.native.container-build=true \
-Dquarkus.openshift.deploy=true \
```

```
-Dquarkus.container-image.name=graalkus-ocp-aot \
-Dquarkus.openshift.name=graalkus-ocp-aot
```