

Springboot 3.0 native images 이동근

목차

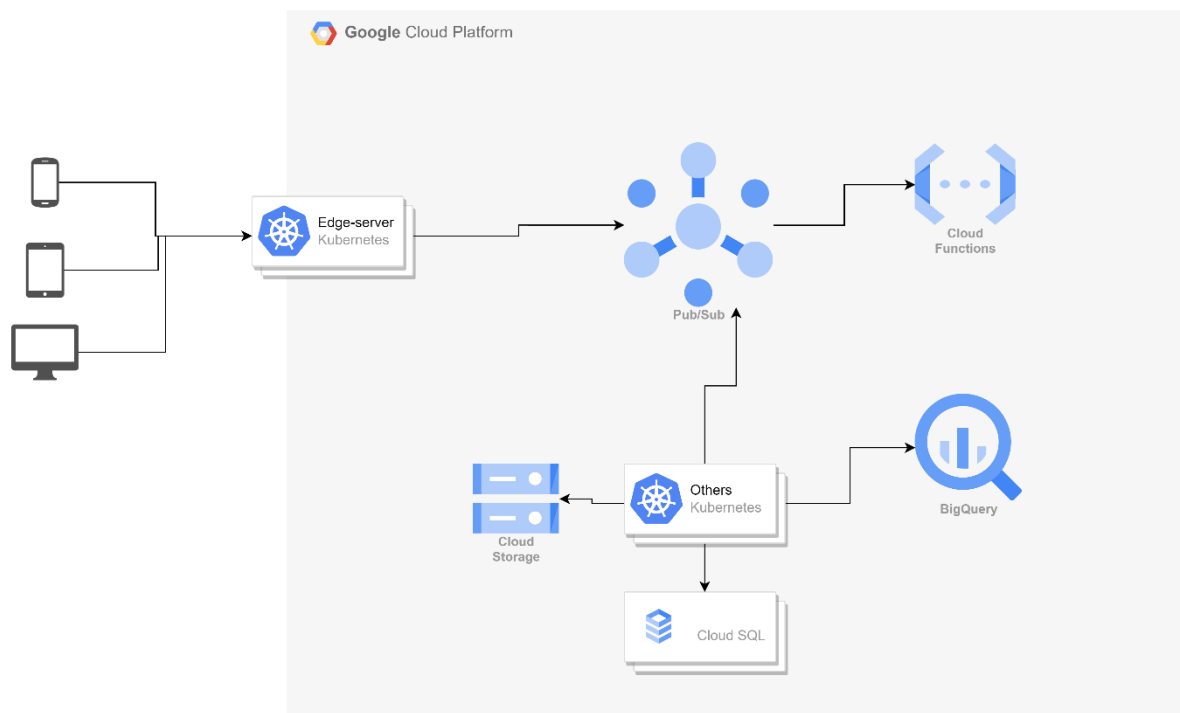
1. 크래시리포트 시스템 구조
2. 문제 정의 및 해소 방안
3. Spring boot 3.0 배포 및 신규 기능
4. Native images 란
 - A. Graalvm JDK
 - B. Ahead of Time / Just in Time compiler
5. Native image 생성하기
 - A. 크래시리포트 서브시스템 중 edge-server 선정
 - B. Spring boot 3 마이그레이션 하기
 - C. Spring-boot:process-aot
6. Native image 실행 후 변경된 사항
7. 성능 테스트 결과
8. Production 적용 후기
9. 문제 해결을 위한 조치 내역 요약
10. 결론
11. 참고자료

1. 크래시리포트 시스템 구조

크래시리포트 시스템은 게임 실행 혹은 예상치 못한 종료 현상 등의 데이터를 수집하여 통계 생성 및 시각화하여 사용자에게 제공하는 시스템이다.

게임에서 발생한 데이터를 전송하기 위해 크래시리포트 SDK는 엣지 서버 클러스터에 접속한다. 쿠버네티스(Kubernetes:K8S)로 구성된 엣지 서버 클러스터는 데이터 인입량이 증가될 경우를 대비해서 Horizontal Pod Autoscaling(HPA) 를 설정하였고 CPU 자원 요청 조건 기준으로 노드 및 파드 (POD)를 증설하고 있다.

Figure 1 크래시리포트 시스템 구조 요약도



2. 문제 정의 및 해소방안

엣지 클러스터는 사용량에 따른 적절한 자원 및 파드 수를 설정하기 어렵다. 게임 사용자가 언제 증가될지 알 수 없는 것이 가장 큰 원인이다. 그래서 HPA 를 사용하여 CPU 자원 요청 비율을 기준으로 파드 수를 조절하고 있다.

하지만, 신규 파드를 추가할 때 최소 1분이상 소요된다. 서버에 접속하는 클라이언트의 통신연결 대기 시간은 대략 15초로 설정되어 있기 때문에 파드가 추가되어 수집할 용량을 늘려 주긴 하지만 신규 파드가 준비되는 시간 동안 누락되는 요청이 많을 것이다.

이를 해결하기 위해서 단순히 15초 이내에 서버를 증설할 수 있는 환경을 만들어야 한다.

3. Spring boot 3.0 배포 및 신규 기능

지난 2022년 11월24일에 스프링부트3.0이 정식 배포*되었다. 최소 지원 Java 의 버전이 17로 변경되었으며, GraalVM 을 사용한 네이티브이미지 생성 기능이 추가되었다.

The General Recursive Applicative and Algorithmic Language Virtual Machine (Graal VM) 은 고성능 JDK 이며 네이티브 이미지 빌더이기도 하다.

네이티브이미지를 생성하는 기능은 2019년 9월부터 진행되던 프로젝트**의 기능을 흡수 통합하여 정식 지원하는 것이다.

*<https://spring.io/blog/2022/11/24/spring-boot-3-0-goes-ga>

**<https://github.com/spring-attic/spring-native>

4. Native images 특징 및 장단점

네이티브이미지는 독립적으로 실행될 수 있게 자바코드를 빌드하는 기술이다.

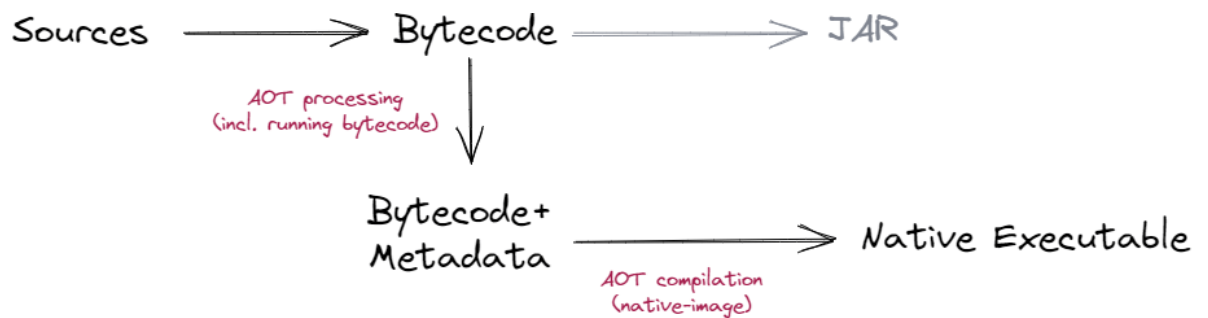
Java Virtual Machine (JVM) 또한 네이티브이미지 내에 포함되기 때문에 실행시 JDK 나 JRE 가 필요하지 않다. 네이티브 이미지는 실행될 플랫폼에 맞춰서 빌드를 해야 된다. 오라클리눅스에서 빌드된 네이티브이미지는 알파인리눅스에서 실행되지 않는다. 이를 위해서 어떤 라이브러리를 사용하여 빌드할지를 정할 수 있다.

네이티브이미지는 실행 환경을 맞춰 빌드가 되기 때문에 컨테이너처럼 제한된 실행 환경에서는 이점이 있다.

네이티브 이미지로 만들어진 프로그램은 서비스를 제공하기 위한 초기 준비 시간이 짧고 메모리 사용량이 적은 장점이 있다.

하지만, 자바코드를 네이티브 이미지로 컴파일 할 때는 Java 의 주요 특징이자 장점인 동적으로 할당하고 실행되던 기능들은 모두 사전 정의하여 컴파일시 참조 할 수 있도록 해야 한다.

Figure 2 Ahead of Time (AOT) / Just in Time (JIT) compiler 수행 절차*



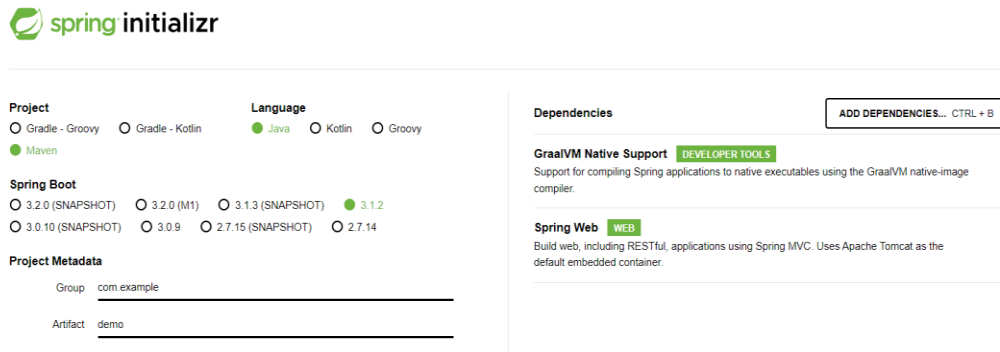
*<https://www.baeldung.com/spring-native-intro>

Figure2 의 언급된 Metadata는 GraalVM 에서 JSON hint files 에 해당 된다.

5. Native image 생성하기

<https://start.spring.io> 에서 GraalVm Native Support 를 프로젝트 추가해주면 된다.

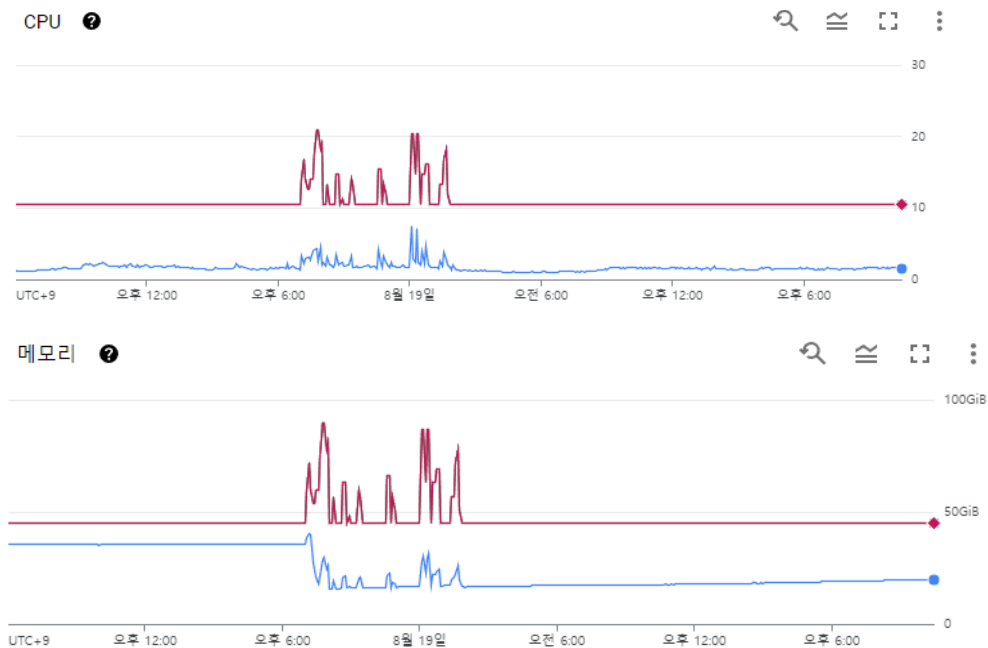
Figure 3 Spring initializr Dependencies 추가 화면



The image shows the Spring Initializr web interface. On the left, under 'Project', 'Maven' is selected. Under 'Language', 'Java' is selected. Under 'Spring Boot', version '3.1.2' is selected. The 'Project Metadata' section shows 'Group' as 'com.example' and 'Artifact' as 'demo'. On the right, the 'Dependencies' section has a button 'ADD DEPENDENCIES... CTRL + B'. Below this, 'GraalVM Native Support' is added as a dependency, with a tag 'DEVELOPER TOOLS'. Below that, 'Spring Web' is added as a dependency, with a tag 'WEB'.

크래시리포트 서브시스템 중 엣지서버는 Horizontal Pod Autoscaling (HPA) 설정이 최소 15에서 최대 30까지 인스턴스를 확장할 수 있으며 오토스케일링 또한 빈번히 발생하고 있다. 다음 그림은 8월18일 금요일 오후 6시 ~ 19일 새벽 3시 사이에 오토스케일링이 적용되어 자원이 늘어나고 줄어듦을 확인 할 수 있다. 하지만 대부분의 시간은 최소 파드로 서비스가 가능한 상태이다. 그렇기 때문에 일시적 최대 요청을 맞추기 위해서 파드의 수를 사전에 더 늘리는 것은 비효율적이다.

Figure 4 HPA 설정에 기반한 자원 할당 변화 차트



위 Figure 4처럼 자원이 증설되는 시간 동안은 클라이언트의 짧은 접속타임아웃 시간으로 인해서 데이터의 유실이 발생한다. 네이티브이미지로 만들기 위해서는 기존의 스프링부트2 프로그램을 스프링부트3으로 마이그레이션*을 먼저 해야한다. 엣지서버의 주요 변경 내용은 다음과 같다.

*<https://github.com/spring-projects/spring-boot/wiki/Spring-Boot-3.0-Migration-Guide>

- Javax.* 를 Jakarta.* 패키지로 변경
- RestTemplate 사용하여 org.apache.httpcomponents.client5:httpclient5 별도 추가
 - A. 기본제공되는 httpclient 라이브러기가 스프링부트3 에서 제외됨.
- Ehcache 사용하여 <classifier>jakarta</classifier> 추가
 - A. Ehcache 는 기존적으로 Javax 패키지로 작성되어 있고 스프링부트3에서 사용하기 위해서는 Jakarta 패키지를 사용한 버전으로 변경해야 한다.

네이티브 이미지 컴파일을 위해서 힌트 파일을 생성해야 한다. 이때 spring-boot-maven-plugin 에서 제공하는 spring-boot:process-aot 를 활용해서 힌트 파일 만든다. 생성 결과는 target/spring-aot 하위에 proxy-config.json / reflect-config.json / resource-config.json / serialization-config.json 파일이 생성된 것을 확인 할 수 있다.

Figure 5 spring-boot:process-aot 실행 결과

```
> mvn site (org.apache.maven.plugins:maven-site-plugin:3.12.1)
> mvn spring-boot (org.springframework.boot:spring-boot-maven-plugin:3.1.2)
  spring-boot:build-image
  spring-boot:build-image-no-fork
  spring-boot:build-info
  spring-boot:help
  spring-boot:process-aot
  spring-boot:process-test-aot
  spring-boot:repackage
  spring-boot:run
  spring-boot:start
  spring-boot:stop
  spring-boot:test-run
> mvn surefire (org.apache.maven.plugins:maven-surefire-plugin:3.0.0)
  > test
  > target
    > classes
    > generated-sources
    > generated-test-sources
    > native
    > spring-aot
      > main
        > classes
        > resources
          > META-INF
            > native-image
              > com.netmarble.meerkat.edge
                > edge-server
                  > native-image.properties
                  > proxy-config.json
                  > reflect-config.json
                  > resource-config.json
                  > serialization-config.json
        > spring
      > sources
```

Spring-boot:process-aot 를 통해서 생성되지 않는 동적 할당 내용들은 사용자 정의 *힌트클래스를 통해서 사전 정의 하면 Spring-boot:process-aot 명령어를 통해서 힌트 파일에 내용이 자동 추가된다.

*<https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#native-image.advanced.custom-hints>

Figure 6. 사용자 정의 힌트 클래스 예시

```
import java.lang.reflect.Method;
import org.springframework.aot.hint.ExecutableMode;
import org.springframework.aot.hint.RuntimeHints;
import org.springframework.aot.hint.RuntimeHintsRegistrar;
import org.springframework.util.ReflectionUtils;

public class MyRuntimeHints implements RuntimeHintsRegistrar {
    @Override
    public void registerHints(RuntimeHints hints, ClassLoader classLoader) {
        // Register method for reflection
        Method method = ReflectionUtils.findMethod(MyClass.class, "sayHello", String.class);
        hints.reflection().registerMethod(method, ExecutableMode.INVOKE);

        // Register resources
        hints.resources().registerPattern("my-resource.txt");

        // Register serialization
        hints.serialization().registerType(MySerializableClass.class);

        // Register proxy
        hints.proxies().registerJdkProxy(MyInterface.class);
    }
}
```


Dockerfile 을 활용한 native image 생성시 기본 이미지를 graalvm-community 버전(ghcr.io/graalvm/graalvm-community:17-ol9)을 사용하고 mvn -Pnative native:compile 명령어로 컴파일 했다. Figure7 과 같이 pom.xml 에 필요 내용을 작성했다.

Figure 7 AOT 컴파일을 하기 위해 Pom.xml 내 정의된 내용

```
<plugin>
  <groupId>org.graalvm.buildtools</groupId>
  <artifactId>native-maven-plugin</artifactId>
  <configuration>
    <mainClass>com.netmarble.meerkat.edge.EdgeServerApplication</mainClass>
    <buildArgs>
      <buildArg>-H:+ReportExceptionStackTraces</buildArg>
      <buildArg>-H:+RunReachabilityHandlersConcurrently</buildArg>
      <buildArg>--trace-class-
initialization=org.apache.commons.logging.LogFactory</buildArg>
      <buildArg>--initialize-at-build-
time=org.apache.commons.logging.LogFactory,org.apache.commons.compress</buildArg>
      <buildArg>-H:DynamicProxyConfigurationFiles=proxy-config.json</buildArg>
      <buildArg>-H:SerializationConfigurationFiles=serialization-
config.json</buildArg>
      <buildArg>-H:ReflectionConfigurationFiles=reflect-config.json</buildArg>
    </buildArgs>
  </configuration>
</plugin>
```

Dockerfile은 알파인리눅스용 네이티브 이미지 만들기 위해 빌드와 실행의 멀티스테이지 환경으로 작성한다.

```
# build stage
FROM ghcr.io/graalvm/graalvm-community:17-ol9 AS builder

# oracle linux 계정 세팅
RUN groupadd appgroup && useradd appuser
RUN usermod -aG appgroup appuser
USER appuser

ARG DOCKER_BUILD_RES=src/main/resources/Dockerfile
ARG USER_HOME=/home/appuser

# musl 설치
WORKDIR $USER_HOME
COPY $DOCKER_BUILD_RES/x86_64-linux-musl-native-10.2.1.tgz $USER_HOME
RUN tar xvfz x86_64-linux-musl-native-10.2.1.tgz
ENV MUSL_HOME=$USER_HOME/x86_64-linux-musl-native

# zlib 설치
WORKDIR $USER_HOME
COPY $DOCKER_BUILD_RES/zlib-1.3.tar.gz $USER_HOME
RUN tar -xvf zlib-1.3.tar.gz
WORKDIR $USER_HOME/zlib-1.3
RUN ./configure --prefix=$MUSL_HOME --static
RUN make
RUN make install

# upx 설치
WORKDIR $USER_HOME
COPY $DOCKER_BUILD_RES/upx-4.1.0-amd64_linux.tar $USER_HOME
RUN tar -xvf upx-4.1.0-amd64_linux.tar
ENV UPX_HOME=$USER_HOME/upx-4.1.0-amd64_linux

# maven 설치
WORKDIR $USER_HOME
ARG MAVEN_VERSION=3.9.4
COPY $DOCKER_BUILD_RES/apache-maven-$MAVEN_VERSION-bin.tar.gz $USER_HOME
```

```

RUN tar -xvf apache-maven-$MAVEN_VERSION-bin.tar.gz
ENV MAVEN_HOME $USER_HOME/apache-maven-$MAVEN_VERSION
ENV MAVEN_CONFIG $USER_HOME/.m2

# SET PATH
ENV PATH=$PATH:$MUSL_HOME/bin:$MAVEN_HOME/bin:$UPX_HOME

# dependency caching
WORKDIR /home/appuser
COPY pom.xml .
RUN mvn -B dependency:resolve
# source 복사 & package
COPY src/main/ src/main/
COPY src/main/resources/proxy-config.json proxy-config.json
COPY src/main/resources/serialization-config.json serialization-config.json
COPY src/main/resources/reflect-config.json reflect-config.json
RUN mvn -Pnative native:compile
RUN upx -7 target/edge-server

```

FROM alpine:3.18

```

# runtime package 설치
#RUN sed 's/https/http/g' -i /etc/apk/repositories
#RUN apk update

# 1-1) alpine 계정 세팅
RUN addgroup -S appgroup && adduser -S appuser -G appgroup
USER appuser

## 1-2) oracle linux 계정 세팅
#RUN groupadd appgroup && useradd appuser
#RUN usermod -aG appgroup appuser
#USER appuser

# 2) Native 실행 파일 복사
COPY --from=builder /home/appuser/target/edge-server /home/appuser

# 3) 변수 설정
ENV SPRING_PROFILES_ACTIVE=${SPRING_PROFILES_ACTIVE:-stg}
EXPOSE 8080

# 4) 실행
WORKDIR /home/appuser
CMD ["/edge-server", "-Xms2g", "-Xmx2g"]

```

6. Native image 실행 후 변경된 사항

실행 문구에 AOT-processed 가 추가 된다.

- Starting AOT-processed EdgeServerApplication using Java 17.0.8 with PID 1

프로그램 준비 완료 시간이 단축됨을 확인할 수 있다.

- Started EdgeServerApplication in 1.905 seconds (process running for 1.911)

Figure 8을 보면 네이티브이미지 파일이 실행될 때 사용되는 CPU 와 메모리가 적게 요청되며 배포가 완료된 것을 확인할 수 있다.

Figure 8. Jar 와 native image 배포 시가 CPU / 메모리 사용을 비교 그래프



7. 성능 테스트

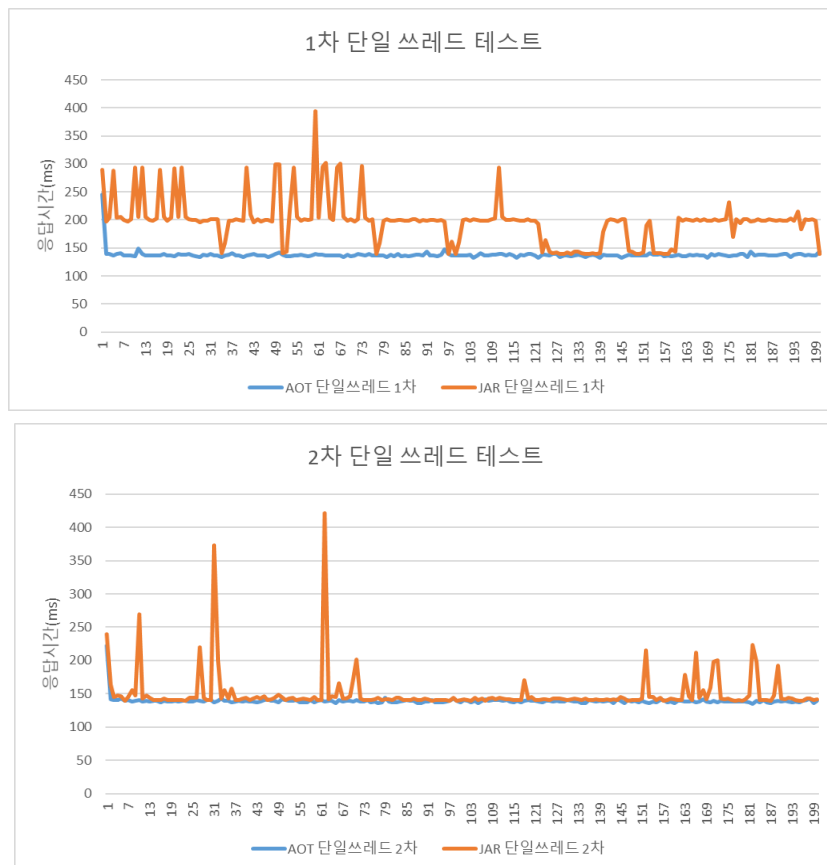
A. 개요

- 테스트 구간 및 소요시간 계산
 1. 클라이언트에서 메시지 전송 후 응답을 받을 때 까지 소요시간(ms)
- 테스트 유형
 1. 단일 쓰레드와 멀티 쓰레드 방식으로 비교 실험함.

B. 결과

- 단일 쓰레드 결과
 1. Figure9 를 통해서 확인한 결과, 둘 다 최초 1회에는 다른 반복 차수 보다 응답시간이 늦었지만, 네이티브 이미지로 실행된 테스트에서 좀 더 안정적인 응답 결과를 얻을 수 있었다.

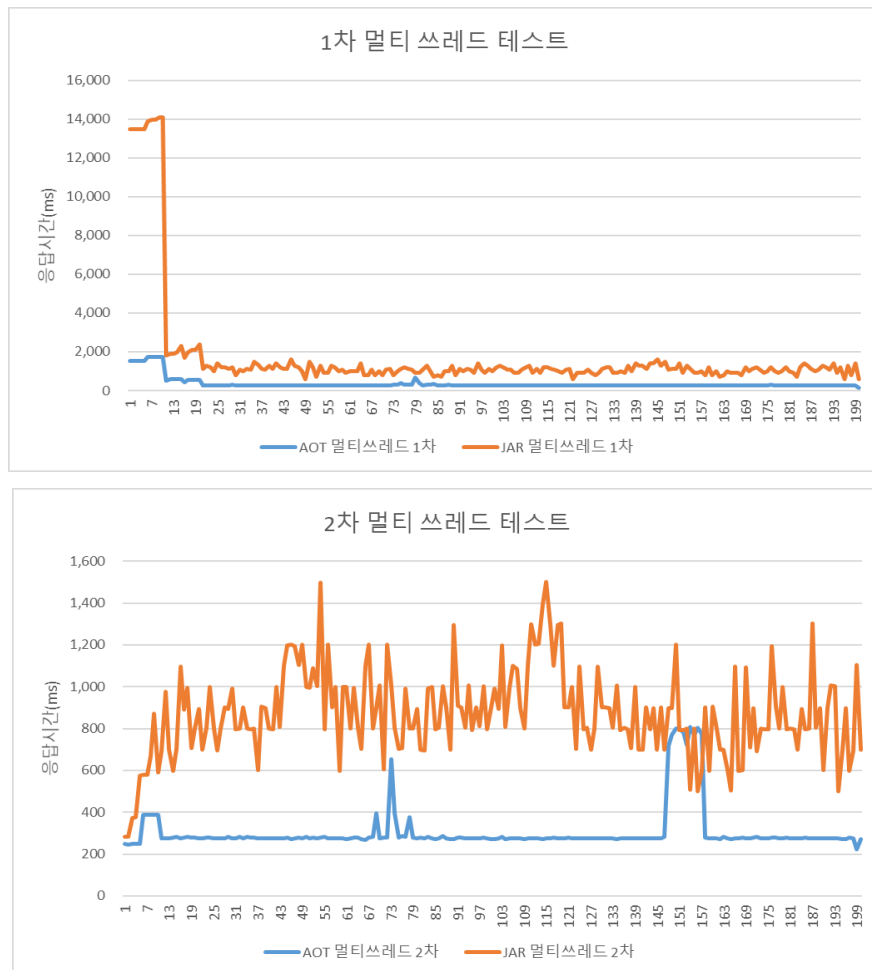
Figure 9. 단일 쓰레드 테스트 결과 그래프



- 멀티 쓰레드 결과

1. Figure 10을 통해서 10개의 쓰레드를 동시에 사용한 테스트 결과, 첫번째 시도시 가장 긴 시간의 응답시간을 나타내었고, 초기 로딩이 끝난이후에는 역시나 네이티브 이미지에서 좀 더 안정적인 응답 성능이 나왔다.

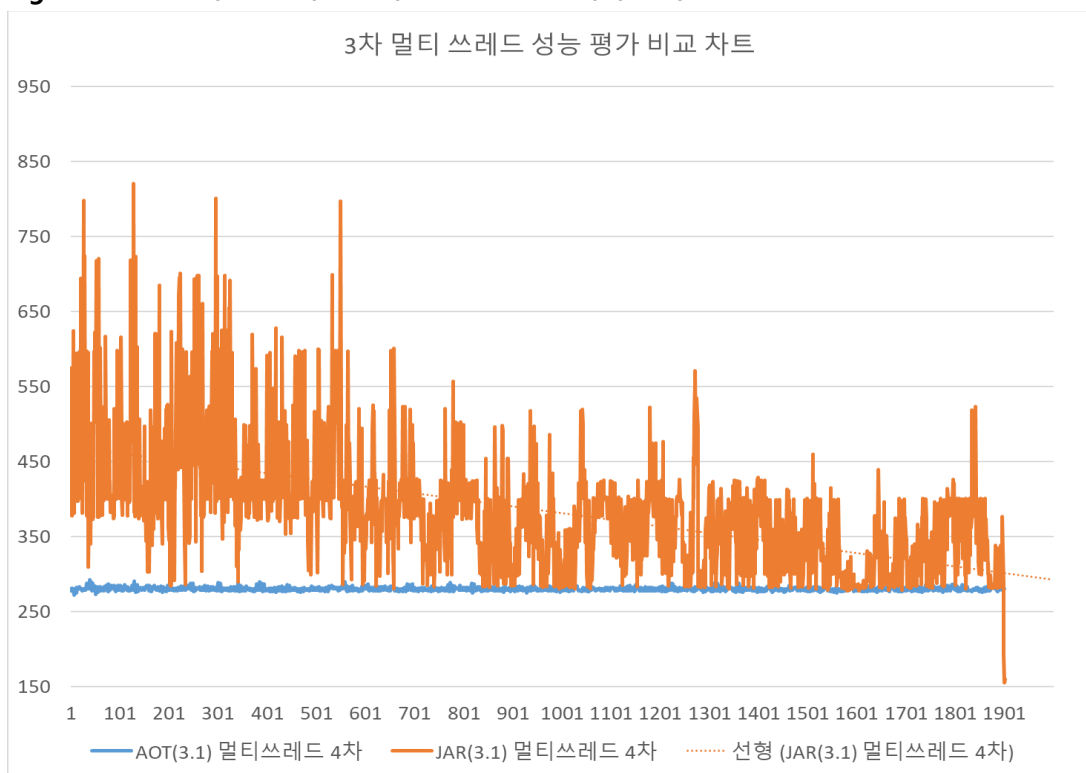
Figure 10. 다중 쓰레드 테스트 결과 그래프



2. Figure 11 은 테스트 횟수를 2000회로 증가한 결과를 나타내면 후반부로 갔을 때 네이티브 이미지와 JVM 기반 Jar 실행 파일의 응답 시간이 250ms 으로 수렴하고 있다. 실제 중심축을 표기하면 Jar실행 파일의 응답 성능이 점점 개선됨을 확인 할 수 있다.

JVM 의 효율적인 캐싱 기능으로 초기 성능차이가 시간이 지남에 따라서 줄어드는 것을 확인할 수 있다. 하지만, 크래시리포트의 엡지서버는 파드가 증설 시간과 초기 응답시간이 중요하기 때문에 이는 참고 자료로만 활용할 뿐이다. 또한 동일 메시지를 2000회 반복 테스트하여 얻은 결과이기 때문에 실제 상황에서는 더 긴 실행 시간이 지난 후에야 성능 역전 현상이 발생할 것으로 예상된다.

Figure 11. 2000회 반복 테스트 비교 및 JAR 중심축의 표기



<https://www.inner-product.com/posts/benchmarking-graalvm-native-image/>

8. Production 적용 후기

이렇게 만들어진 edge-server 네이티브 이미지는 다섯 차례에 걸쳐서 배포되었고 그 과정에서 네이티브 이미지 특성에 대해서 더 알게 되었다.

A. 1차 8월 22일 배포

- 빌드 환경

1. FROM vegardit/graalvm-maven:latest-java17 AS builder, Debian-slim 기반 이미지

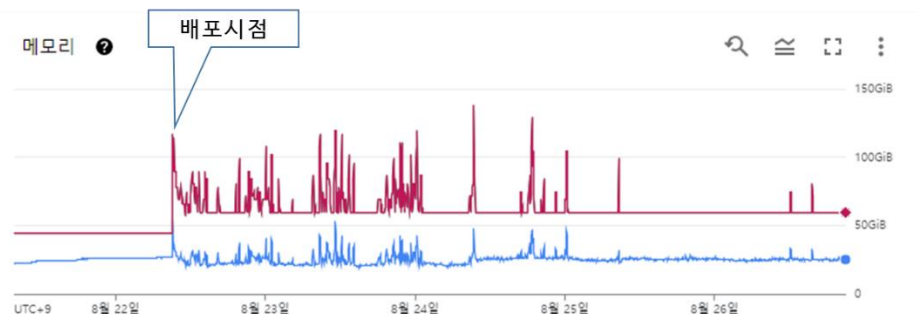
- 실행 환경

1. FROM eclipse-temurin:17-jre-jammy AS runner, Ubuntu 기반 이미지

빌드 환경과 실행 환경의 운영체제가 달라서인지 실행시 CPU 사용율이 높았고 Out of Memory(OOM)으로 인한 재 시작이 자주 발생했다. CPU 사용율의 진폭이 넓어 파드(pod) 수의 증감이 빈번히 발생했다. 이를 대응하고자 할당된 CPU 자원을 800m에서 1200m으로 변경하였다. 최소 파드 수를 15 에서 20으로 변경하고 파드 증가 조건인 CPU 자원 요청율을 50%에서 60%로 조정하여 1차 배포를 대응 했다.

1차 배포 때는 기능적으로는 문제가 없이 실행되지만 안정적으로 작동할 것이라고 보장하기 어려웠고 배포 후 결과를 봐도 기존 JAR 대비 불안정하게 작동하고 있었다.

Figure 12. 1차 배포 후 리소스 사용량에 대한 그래프



B. 2차 8월 25일 배포

- 빌드 환경

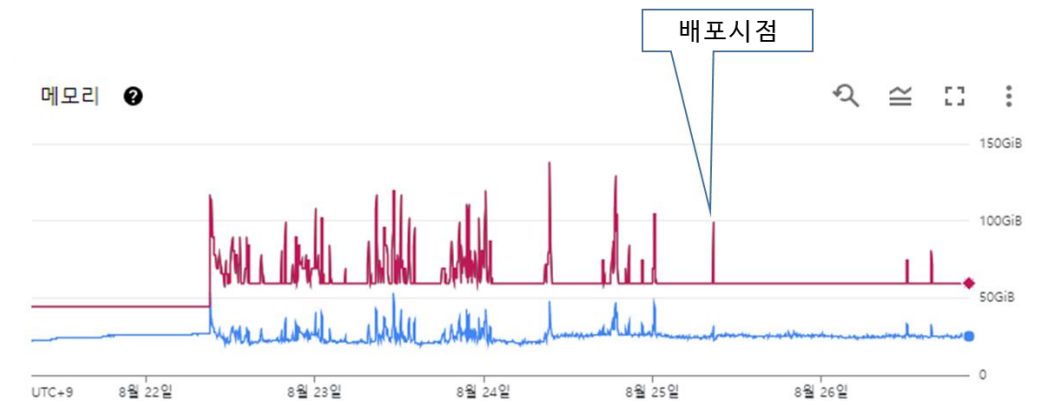
1. FROM ghcr.io/graalvm/graalvm-community:17-ol9 AS builder, Oracle Linux 9 기반 이미지

- 실행 환경

1. FROM ghcr.io/graalvm/jdk-community:latest AS runner, Oracle Linux 9 기반 이미지

빌드환경과 실행환경을 일치하면서 이전 대비 OOM 으로 인한 재시작이 2-3 회로 줄어들고 발생하지 않은 파드가 대부분이며 증감도 줄어들었다.

Figure 13. 2차 배포 후 리소스 사용량에 대한 그래프



C. 3차 8월 28일 배포

- 빌드 환경

1. FROM ghcr.io/graalvm/graalvm-community:17-ol9 as builder, Oracle Linux 9 기반 이미지

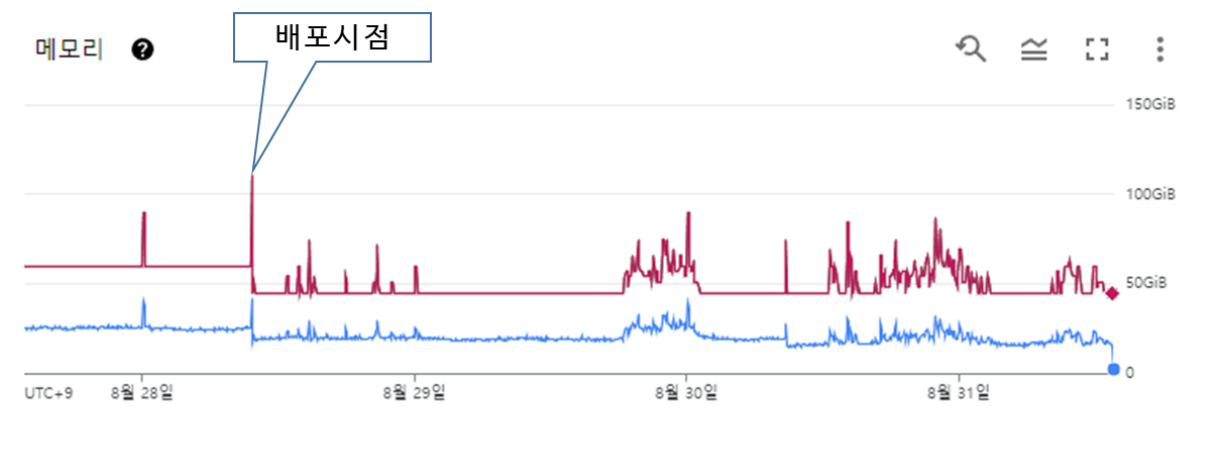
- 실행 환경

1. FROM ghcr.io/graalvm/jdk-community:17-ol9 as runner, Oracle Linux 9 기반 이미지

그동안 수정된 edge-server 내역을 반영하기 위해서 spring-boot:process-aot 재실행 후 변경 사항을 적용(reflect-config.json 변경) 하였으며, 네이티브 이미지를 위한 tomcat 버전(tomcat-embed-programmatic) 을 적용하였다. 최소 파드수를 20 에서 15 로 낮춰 1차 배포 이전 값으로 원복 했다. 이로써 최초 네이티브 이미지 배포 전과 동일한 파드 수가 되어서 비교 분석할 수 있게 되었다. Graalvm 은 Serial GC 옵션을 기본으로 제공하고 *G1 GC 옵션을 사용하기 위해서는 Enterprise 유료버전을 사용해야 한다. Serial GC 를 사용하기 때문에 GC에 필요한 메모리가 G1 보다 더 많이 필요하며 이로 인해 OOM 은 간혹 발생할 것으로 예상된다. 그래서 실제 메모리 할당을 조금 더 많이 해줘야 한다.

*<https://www.graalvm.org/latest/reference-manual/native-image/optimizations-and-performance/MemoryManagement/#g1-garbage-collector>

Figure 14. 3차 배포 후 리소스 사용량에 대한 그래프



D. 4차 8월 30일 배포

- 빌드 환경

1. FROM ghcr.io/graalvm/graalvm-community:17-ol9 AS builder, Oracle Linux 9 기반 이미지

- 실행 환경

1. FROM alpine:3.18 AS runner, Alpine Linux 기반 이미지

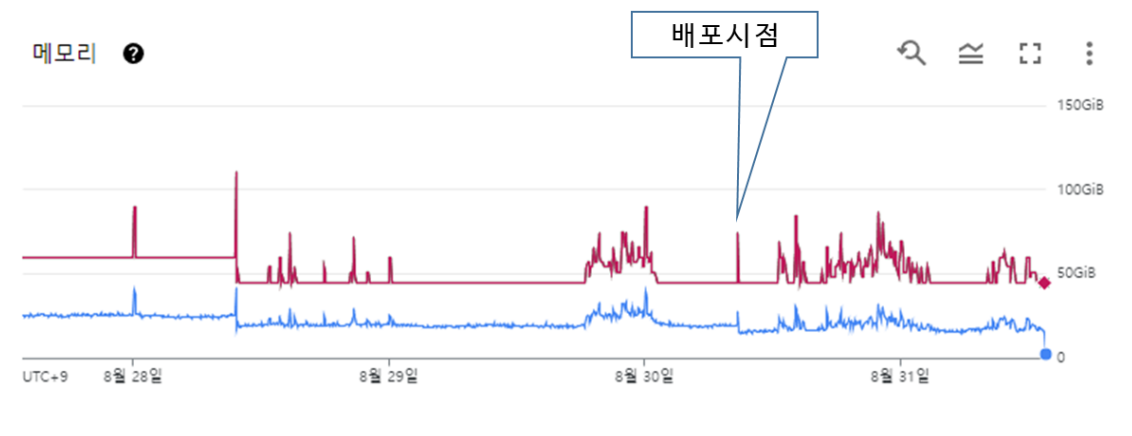
4차 배포의 특징은 tomcat-embed-programmatic 버전을 10.1.1에서 10.1.11로 업그레이드를 하고 CPU 와 메모리 할당 설정을 기존 JAR 시절처럼 원복을 한 것이다. 이로써 기존 JAR 시절과 동일한 자원으로 서비스를 할 수 있게 되었다.

4차 배포부터는 빌드시 `-libc:musl` 옵션**을 추가해서 Alpine Linux 에서 실행*되도록 빌드를 했다. 기존의 Oracle Linux 대비 Alpine Linux 가 용량도 적고 컨테이너 환경에서는 더 나은 선택이라고 생각했다.

*<https://www.graalvm.org/latest/reference-manual/native-image/guides/build-static-executables/>

**<https://www.graalvm.org/latest/reference-manual/native-image/overview/BuildOptions/>

Figure 15. 4차 배포 후 리소스 사용량에 대한 그래프



E. 5차 9월 1일 배포

- 빌드 환경

1. FROM ghcr.io/graalvm/graalvm-community:17-ol9 (oracle linux 9) AS builder

- 실행 환경

1. FROM alpine:3.18 AS runner

5차 배포의 특징은 the Ultimate Packer for eXecutables(UPX) 적용*한 것이다. UPX 는 네이티브이미지로 생성된 실행파일의 크기를 줄여주는 도구다. 이를 활용하면 컨테이너의 크기를 줄여주는데 큰 도움이 된다. 실제로 Alpine Linux 는 10MB 내외지만, 실행 파일은 100MB에 다다랐으며, 이를 대략 60MB까지 줄여줬다. 그 외 실행 환경에서 불필요한 파일들을 제거함으로써 최종적으로 도커 이미지 크기는 69.81MB까지 감소했다. 시간 순서로 도커 이미지 크기를 정리하면 다음과 같다.

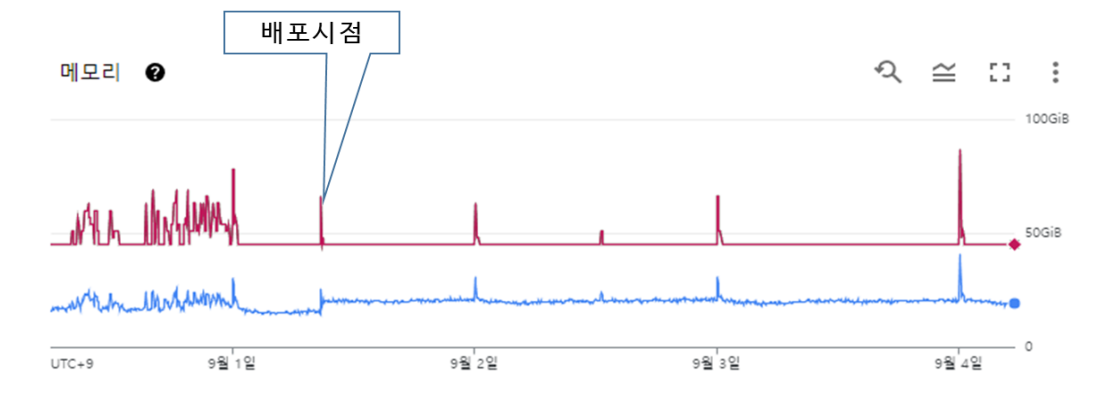
- 2차 8월 25일 배포 985.55 MB
- 4차 8월 30일 배포 344.33 MB
- 5차 9월 1일 배포 69.81 MB

쿠버네티스는 해당 파드가 서비스할 준비(readinessProbe)가 되었는지 혹은 서비스 중(livenessProbe)인지를 체크하는 기능이 있다. 이 옵션을 5차 배포 때 추가 되었다. 네이티브이미지로 실행되는 서버가 2초 내외로 준비가 되면 이를 체크해서 최대 5초 내외로 클라이언트의 요청이 파드 내로 신속하게 전달되게 하기 위함이다.

이번 5차 배포 후 OOM 은 특정 파드에서 일 1회 미만으로 발생하거나 발생하지 않았다. 리소스 사용량 그래프를 보면 주말인 날짜를 감안하더라도 상당히 안정적인 자원 사용을 확인할 수 있다.

*<https://medium.com/graalvm/compressed-graalvm-native-images-4d233766a214>

Figure 16. 5차 배포 후 리소스 사용량에 대한 그래프



9. 문제 해결을 위한 조치 내역 요약

초기 실행시간을 줄이기 위해서 SpringBoot 3에서 지원하는 Native 이미지를 배포 파일을 만들어서 기존 JAR 실행시간 50초에서 2초로 줄어들었습니다. 신규 파드가 추가될 때 사용되는 토커 이미지 크기를 최적화되어 기존 JAR 기반 도커 이미지 크기가 300MB에서 70MB 로 줄어들었다.

10. 결론

9월1일 배포이후 엣지 서버의 서비스가 안정으로 제공되고 있습니다. 줄어든 도커 이미지 크기와 실행 시간이 향후 트래픽이 일시적으로 증가할 때 이전 보다 더 빠른 확장으로 안정적인 서비스가 제공될 것으로 예상된다.

네이티브 이미지로 만들어서 서비스하는 것이 무조건 좋은 것은 아니다. 실제로 2000회 이상 반복 테스트를 하게 되면 JAR 를 사용한 서비스가 더 안정적인 성능을 제공하는 것으로 나온 테스트 결과도 있다. 하지만, 쿠버네티스 환경에서 파드의 증설이 빈번히 일어나고 파드의 서비스 준비 시간이 짧아야 된다면 그리고 초기 응답을 위한 로딩 시간을 줄여야 한다면, 이 때는 충분한 고려 대상이 될 수 있다고 생각한다.

11. 참고자료

- A. <https://spring.io/blog/2022/11/24/spring-boot-3-0-goes-ga>
- B. <https://docs.spring.io/spring-boot/docs/current/reference/html/native-image.html#native-image.introducing-graalvm-native-images>
- C. <https://www.baeldung.com/spring-native-intro>
- D. <https://amrutprabhu.medium.com/building-native-image-for-a-spring-boot-application-b9df25556120>
- E. <https://github.com/spring-attic/spring-native>
- F. <https://mangkyu.tistory.com/302>
- G. <https://docs.spring.io/spring-boot/docs/current/reference/html/native-image.html#native-image.advanced.known-limitations>
- H. <https://upx.github.io/>