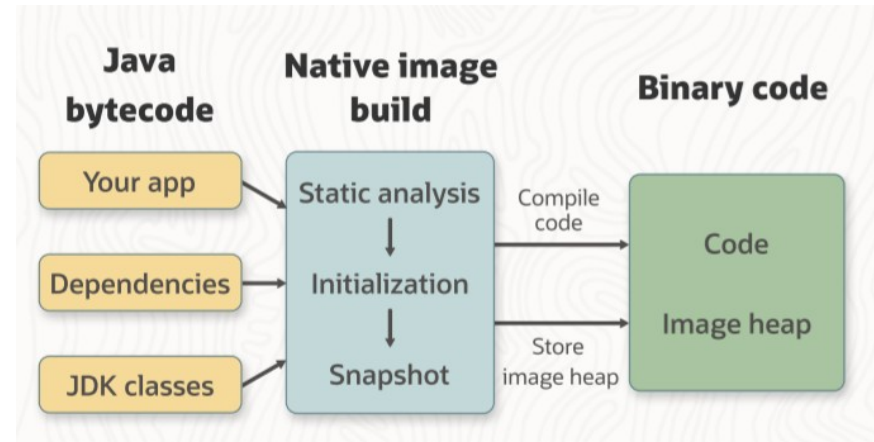


Spring Native

Jiří Pinkas
twitter: @jirkapinkas
<https://github.com/jirkapinkas>

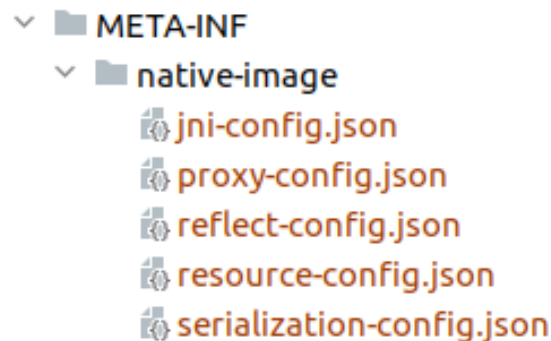
GraalVM Native Image

- Native Image je technologie, která zkompiluje ahead-of-time Java kód do standalone spustitelné aplikace nazvané “native image”. Tato aplikace obsahuje aplikační třídy, třídy co se používají z dependencies, třídy co se používají z Java runtime a native kód z JVM.
- Native image neběží na JVM, ale obsahuje důležité JVM komponenty jako memory management, thread scheduling atd. z jiného runtime, který má název “Substrate VM”. Výsledná aplikace má výrazně rychlejší start a používá méně RAM než JVM.



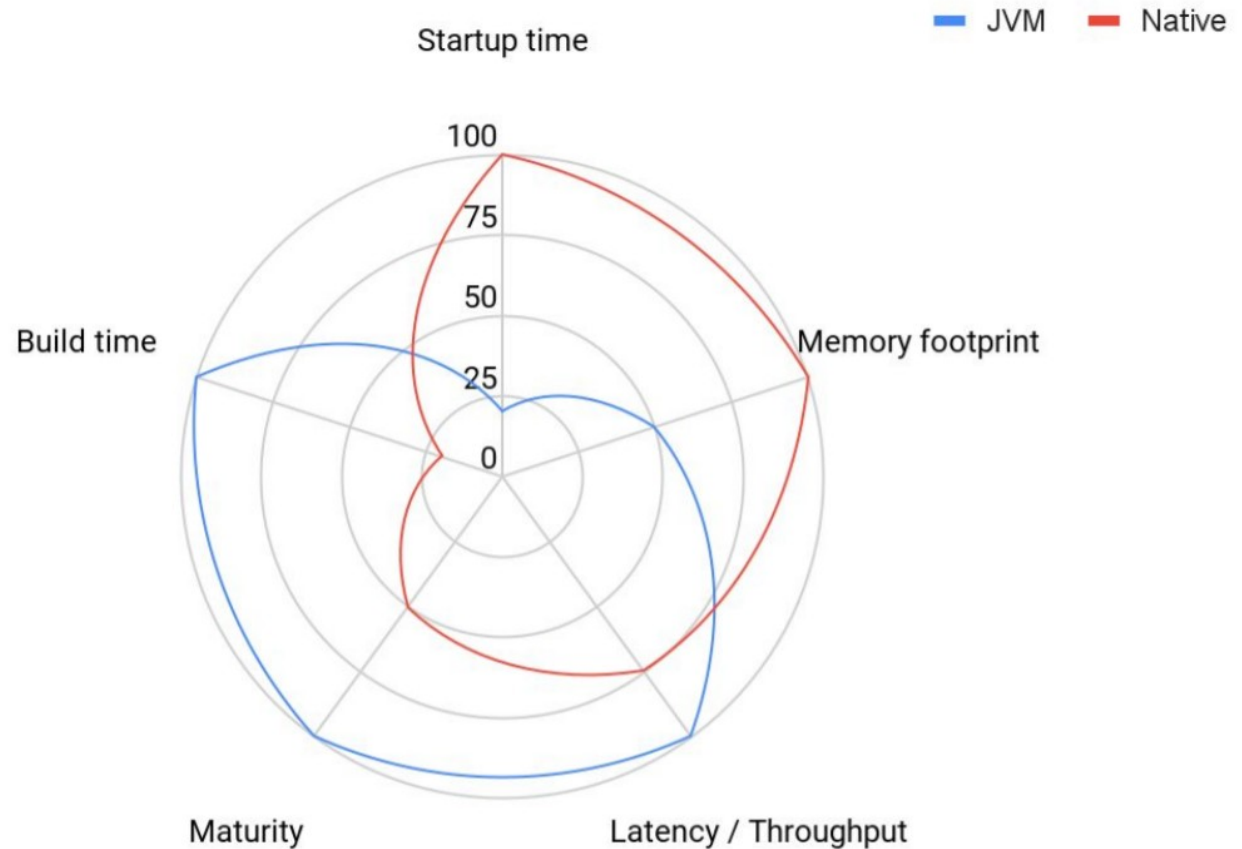
Native & dynamický kód

- Jaké třídy se uloží do Native Image záleží na výsledku statické analýzy kódu při buildu Native Image. Tato analýza ze své podstaty nemůže zjistit, že se za běhu aplikace budou používat JNI (Java Native Interface), reflexe, dynamické proxy nebo resources z classpath.
- Tyto třídy a soubory je zapotřebí při buildu přidat do speciálních souborů (jni-config.json, reflect-config.json, proxy-config.json & resource-config.json). Abychom to nemuseli dělat ručně, Spring Native drtivou většinu práce provede za nás. A samozřejmě je možné i se Spring Native tyto soubory ručně používat.
- GraalVM out-of-the-box neumožňuje dynamicky vytvářet nové třídy v runtime!



JVM vs. Native

- Všechno má své výhody a nevýhody:



Use Cases

- Microservices
 - Menší Docker image, rychlejší start a zejména výrazné snížení vytížení paměti!
- Serverless
 - Java aplikace nespustí X sekund, ale desítky až stovky milisekund!

Jak na build native aplikace

- Buildpacks

- Pomocí Spring Boot Maven pluginu
- Vyžaduje Docker, ale nevyžaduje GraalVM native-image kompilátor
- Výsledkem je Docker image
 - `mvn spring-boot:build-image`

Tady používám
<https://sdkman.io/>

- Native Build Tools

- <https://graalvm.github.io/native-build-tools/>
- Vhodné pro testování
- Vyžaduje nainstalovaný GraalVM native-image kompilátor
- Výsledkem je native executable
 - `sdk i java 22.1.0.r17-grl`
 - `sdk i maven`
 - `gu install native-image`
 - `mvn -Pnative -DskipTests package`

Native Hints

// native-image option or trigger for other hints -> native-image.properties

@NativeHint(options = "--enable-https")



Toto je také možné dát do:

<image><env><BP_NATIVE_IMAGE_BUILD_ARGUMENTS>
(konfigurace spring-boot-maven-plugin s Buidpack podporou)

// Reflection -> reflect-config.json ; *.class resources -> resource-config.json

@TypeHint(types = Data.class, typeNames = "com.example.webclient.Data\$SuperHero")



Třídy, které se používají přes reflexi

// Additional resource to include in the native executable -> resource-config.json

@ResourceHint(patterns = "custom.sql")



Do native aplikace se automaticky kopíruje vše co je v
src/main/resources a src/test/resources.
Tímto tam dokážete nakopírovat i něco jiného.

// JDK dynamic proxy on interfaces -> proxy-config.json

@JdkProxyHint(types = { org.springframework.context.annotation.Lazy.class,
org.springframework.core.annotation.SynthesizedAnnotation.class})

// Class with serialization support -> serialization-config.json

@SerializationHint(types = org.springframework.samples.petclinic.model.Person.class)

// Used to configure which class should be initialized at build-time -> native-image.properties

@InitializationHint(types = com.google.protobuf.Extension.class,
initTime = InitializationTime.BUILD)

Ahead-of-time proxy

```
// Typical security use case of a class proxy now supported on native
@Service
public class GreetingService {

    public String hello() {
        return "Hello!";
    }

    @PreAuthorize("hasRole('ADMIN')")
    public String adminHello() {
        return "Goodbye!";
    }
}
```

```
// Hint for Build-time proxy on classes
-> Spring AOT plugin
@AotProxyHint(targetClass =
    org.springframework.batch.core.launch.su
pport.SimpleJobOperator.class,
proxyFeatures = ProxyBits.IS_STATIC)
```



Native image neumožňuje vytváření nových tříd v runtime. Díky tomuto hintu se proxy vygeneruje při buildu aplikace a v runtime se jenom použije.

JdkProxyHint vs AotProxyHint

- V případě, že naše Spring beana (v tomto příkladu NewsService) implementuje interface a vyžaduje tvorbu proxy (například kvůli @Transactional anotaci), pak se dá použít JdkProxyHint:

```
@JdkProxyHint(types = {  
    cz.jiripinkas.skoleni.service.NewsService.class,  
    org.springframework.aop.SpringProxy.class,  
    org.springframework.aop.framework.Advised.class,  
    org.springframework.core.DecoratingProxy.class  
})
```

- Pokud interface neimplementuje, pak se musí použít AotProxyHint:

```
@AotProxyHint(targetClass=cz.jiripinkas.skoleni.service.NewsService.class,  
    proxyFeatures = ProxyBits.IS_STATIC)
```

Konfigurace

- Je dobré mít Native konfiguraci v samostatné @Configuration třídě, dostal jsem se do divné situace když jsem kombinoval @TypeHint a @EnableCaching v jedné třídě, čili je dobré vytvořit něco takového:

```
@NativeHint(options = "--enable-https -H:+AddAllCharsets")
@TypeHint(types = {
    Properties.class,
    Property.class,
    NewsList.class,
    NewsItem.class,
    NewsDetail.class,
    CurrentNews.class
}, access = {TypeAccess.DECLARED_METHODS, TypeAccess.DECLARED_CONSTRUCTORS})
@AotProxyHint(targetClass=cz.jiripinkas.skoleni.service.NewsService.class, proxyFeatures = ProxyBits.IS_STATIC)
@AotProxyHint(targetClass=cz.jiripinkas.skoleni.repo.IntranetRepository.class, proxyFeatures = ProxyBits.IS_STATIC)
@Configuration
public class NativeConfiguration {
}
```

- Když při startu aplikace neuvidíte stacktrace, ale nic-neříkající info že máte něco špatně, tak spusťte aplikaci s --debug=true, pak ten stacktrace uvidíte.

Konfigurace

- Stejná konfigurace jako na předcházejícím snímku se dá zapsat také tímto způsobem:

```
@NativeHint(options = "--enable-https -H:+AddAllCharsets",
    types = {
        @TypeHint(types = {
            Properties.class,
            Property.class,
            NewsList.class,
            NewsItem.class,
            NewsDetail.class,
            CurrentNews.class
        }, access = {TypeAccess.DECLARED_METHODS, TypeAccess.DECLARED_CONSTRUCTORS})
    }, aotProxies = {
        @AotProxyHint(targetClass = cz.jiripinkas.skoleni.service.NewsService.class, proxyFeatures = ProxyBits.IS_STATIC),
        @AotProxyHint(targetClass = cz.jiripinkas.skoleni.repository.IntranetRepository.class, proxyFeatures = ProxyBits.IS_STATIC)
    })
@Configuration
public class NativeConfiguration {
}
```

Rozdíl je v tom, že na `@NativeHint` je atribut `trigger`, pomocí kterého se dá tato konfigurace zapojit při přítomnosti nějaké třídy v classpath.

Konfigurace

- Konfigurace se dá zapsat také programově.
- pom.xml:

```
<dependency>
  <groupId>org.springframework.experimental</groupId>
  <artifactId>spring-aot</artifactId>
  <version>${spring-native.version}</version>
</dependency>
<dependency>
  <groupId>org.reflections</groupId>
  <artifactId>reflections</artifactId>
  <version>0.10.2</version>
</dependency>
```

- src/main/resources/META-INF/spring.factories:

```
org.springframework.aot.context.bootstrap.generator.infrastructure.nativex.BeanFactoryNativeConfigurationProcessor=\
  cz.jiripinkas.skoleni.CustomClientNativeConfigurationProcessor
```

Konfigurace

- Custom anotace:

```
public @interface Dto {  
}
```

- DTOčka vypadají následovně:

```
@Dto  
public record Property (  
    String key,  
    String value  
) {  
}
```

Konfigurace

```
@NativeHint(options = "--enable-https -H:+AddAllCharsets")
@AotProxyHint(targetClass=cz.jiripinkas.skoleni.service.NewsService.class, proxyFeatures = ProxyBits.IS_STATIC)
@AotProxyHint(targetClass=cz.jiripinkas.skoleni.repository.IntranetRepository.class, proxyFeatures = ProxyBits.IS_STATIC)
@Configuration
@Slf4j
public class CustomClientNativeConfigurationProcessor implements BeanFactoryNativeConfigurationProcessor {

    public void process(ConfigurableListableBeanFactory beanFactory, NativeConfigurationRegistry registry) {
        var rootPackage = SkoleniWebApplication.class.getPackageName();
        log.info("scan for DTOs in root package (and it's subpackages): {}", rootPackage);
        var classesSet = new Reflections(rootPackage).getTypesAnnotatedWith(Dto.class);
        log.info("registered DTOs: {}", classesSet);
        classesSet.forEach(aClass -> {
            registry.reflection().forType(aClass)
                .withAccess(TypeAccess.DECLARED_CONSTRUCTORS, TypeAccess.DECLARED_METHODS);
        });
    }
}
```

@Configuration & @Bean

- U Spring Configuration tříd je možné přímé volání jednotlivých @Bean metod. Spring vytvoří CGLIB proxy, pomocí které tato “magie” funguje. Se Spring Native je toto problém, proto Spring tým nově říká, že bychom toto neměli používat a místo toho bychom měli preferovat autowiring přes argumenty. Dá se to také vyloženě zakázat pomocí: @Configuration(proxyBeanMethods = false)

BAD:

```
@Configuration
public class JdbcConfiguration {

    @Bean
    public DataSource dataSource() {
        System.out.println("data source constructed!");
        HikariDataSource ds = new HikariDataSource();
        ds.setJdbcUrl("jdbc:hsqldb:hsqldb://localhost/eshop");
        ds.setUsername("sa");
        ds.setPassword("");
        return ds;
    }

    @Bean
    public JdbcTemplate jdbcTemplate() {
        return new JdbcTemplate(dataSource());
    }
}
```

GOOD:

```
@Configuration
public class JdbcConfiguration {

    @Bean
    public DataSource dataSource() {
        System.out.println("data source constructed!");
        HikariDataSource ds = new HikariDataSource();
        ds.setJdbcUrl("jdbc:hsqldb:hsqldb://localhost/eshop");
        ds.setUsername("sa");
        ds.setPassword("");
        return ds;
    }

    @Bean
    public JdbcTemplate jdbcTemplate(DataSource dataSource) {
        return new JdbcTemplate(dataSource);
    }
}
```

Argumenty aplikace

- Spring Boot aplikaci se dají klasickým způsobem předávat argumenty:
 - `docker run --rm -it -p 8082:8082 TODO_IMG --server.port=8082`
- GraalVM Native to, co jsou v klasické Javě parametry VM bere z argumentů:
 - `docker run -m 200m --rm -it -p 8080:8080 IMG_NAME -XX:+PrintGC -XX:+VerboseGC`

Spring Boot Maven Plugin

- Build aplikace se typicky provádí pomocí Spring Boot Maven Pluginu pomocí:
 - `mvn spring-boot:build-image`
- Dokumentace:
 - <https://docs.spring.io/spring-boot/docs/current/maven-plugin/reference/htmlsingle/>
- Nejpoužívanější konfigurace buildpacků:
 - <https://github.com/paketo-buildpacks/native-image>
 - <https://github.com/paketo-buildpacks/bellsoft-liberica>
 - <https://github.com/paketo-buildpacks/maven>

Reflexe

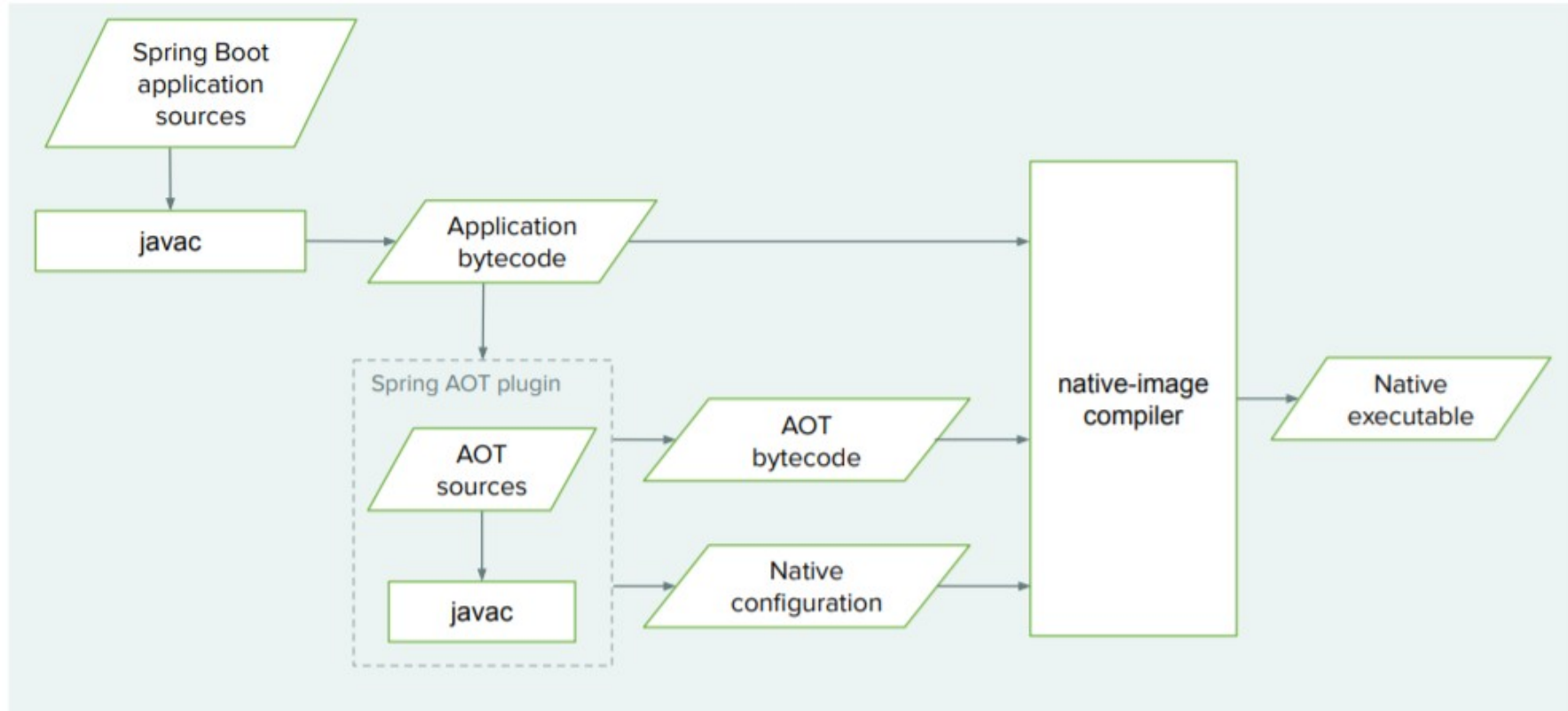
- Celá řada frameworků používá reflexi a doposud jsme nad tím nemuseli moc přemýšlet:
 - @Value používá SpEL (Spring Expression Language)
 - Jackson (RestTemplate, WebClient)
 - Dozer / Orika
 - Thymeleaf (pokud tohle ještě někdo používá :-))
 - ...
- Pro jakoukoli třídu, kterou použijete přes reflexi musíte přidat následující anotaci:
 - @TypeHint(types=Properties.class, access=TypeAccess.AUTO_DETECT)

UPX compression (od 0.11)

- Od 0.11 Buildpacky umí UPX kompresi
 - Až 4x zmenšení image!
 - Sice se prodlouží build, ale ten stejně dlouho trvá.
 - Startup je dokonce o malinko rychlejší, protože dekomprese je rychlejší než čtení větší image z disku :-)

```
<plugin>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-maven-plugin</artifactId>
  <configuration>
    <image>
      <builder>paketobuildpacks/builder:tiny</builder>
      <env>
        <BP_NATIVE_IMAGE>true</BP_NATIVE_IMAGE>
        <BP_BINARY_COMPRESSION_METHOD>upx</BP_BINARY_COMPRESSION_METHOD>
      </env>
    </image>
  </configuration>
</plugin>
```

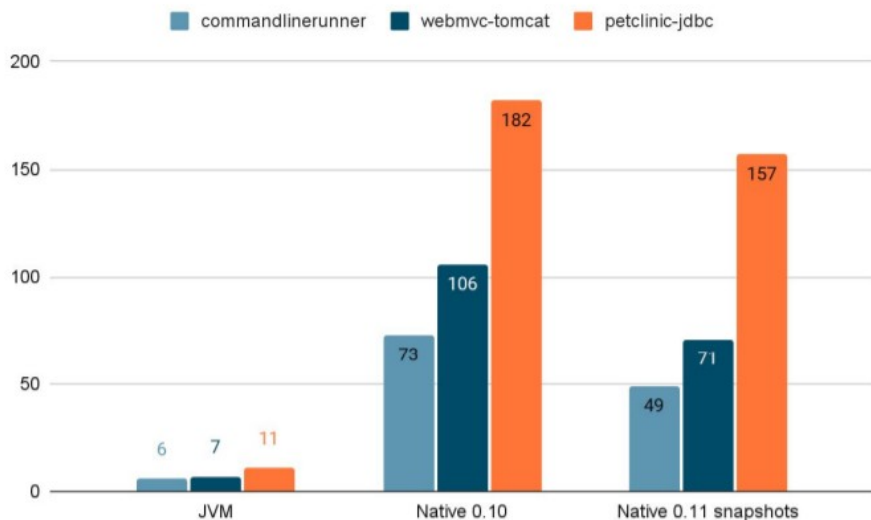
Kompilace Spring Boot aplikace do Native



AOT

- Spring AOT generuje reflect-config.json a další json soubory, které používá GraalVM do adresáře:
 - target/generated-sources/spring-aot/src/main/resources/META-INF/native-image/org/springframework/aot/spring-aot
 - Užitečné, když se aplikace chová divně a máte podezření například, že problém je v nesprávné konfiguraci reflexe.

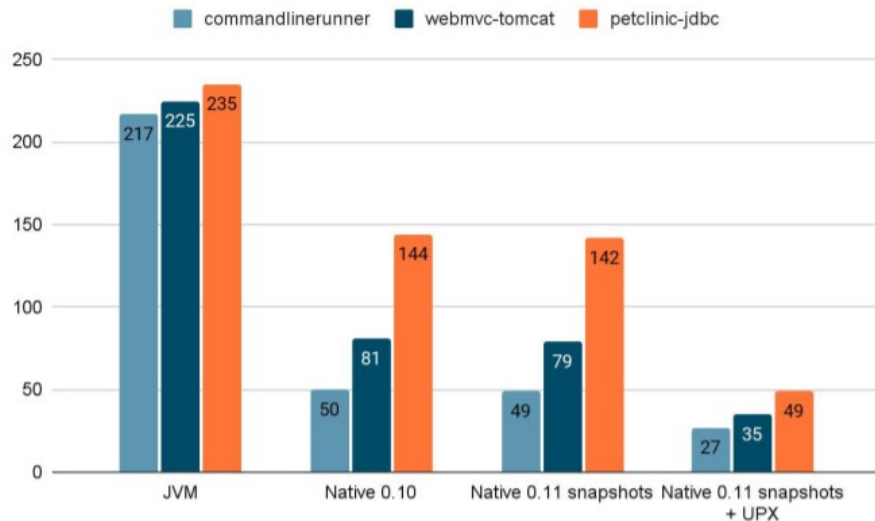
Build time



Details

- Build time in seconds
- Spring Boot 2.5
- GraalVM 21.1 for 0.10, GraalVM 21.2 for 0.11
- Java 11 on Linux
- Laptop, Intel Core i7 8850H @ 2.6 GHz, 32G RAM

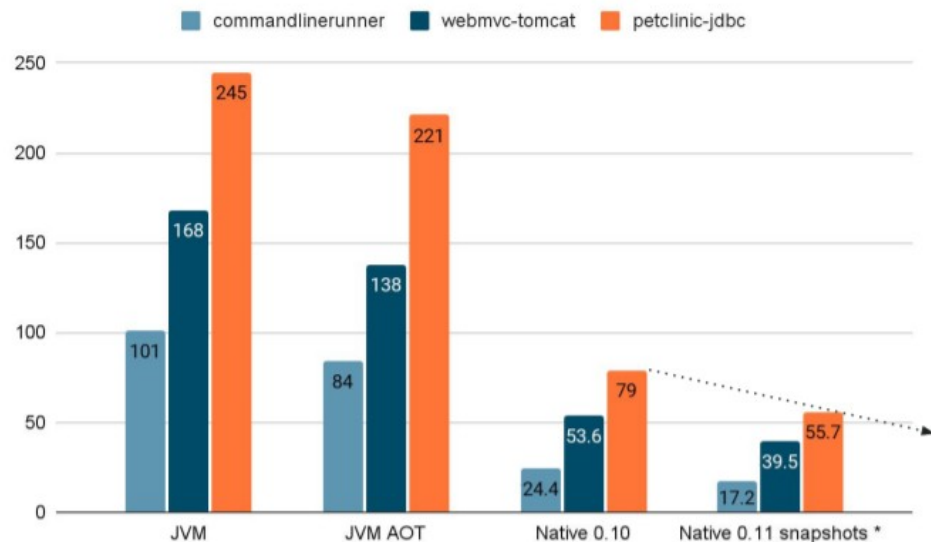
Image size



Details

- Container image size in Megabytes
- Spring Boot 2.5
- Liberica JDK 11.0.12 for JVM, GraalVM CE 21.1 for 0.10, GraalVM CE 21.2 for 0.11
- Java 11 on Linux

Memory footprint

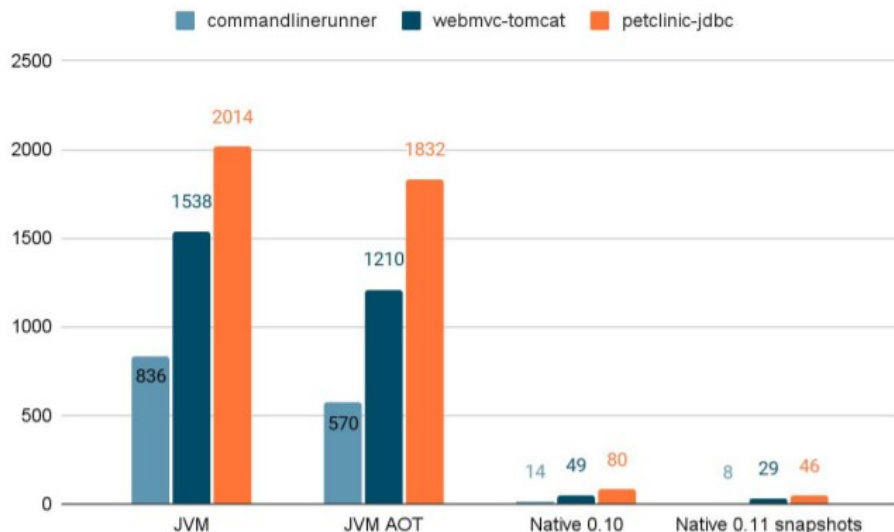


* Expect lower memory consumption in 0.11.0 with GraalVM 21.3 optimizations on reflection

Details

- RSS memory footprint after startup in Megabytes
- Spring Boot 2.5
- Liberica JDK 11.0.12 for JVM, GraalVM CE 21.1 for 0.10, GraalVM CE 21.2 for 0.11
- JVM application ran as an unpacked archive
- Java 11 on Mac

Startup time




Details

- Startup time (including the JVM) in milliseconds
- Simple web application with Spring MVC + Tomcat
- Spring Boot 2.5
- Liberica JDK 11.0.12 for JVM, GraalVM CE 21.1 for 0.10, GraalVM CE 21.2 for 0.11
- JVM application ran as an unpacked archive
- Java 11 on Linux
- Laptop, Intel Core i7 8850H @ 2.6 GHz, 32G RAM

Build & Push do Docker registry

```
<plugin>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-maven-plugin</artifactId>
  <configuration>
    <classifier>${repackage.classifier}</classifier>
    <image>
      <builder>paketobuildpacks/builder:tiny</builder>
      <env>
        <BP_NATIVE_IMAGE>true</BP_NATIVE_IMAGE>
      </env>
      <name>TODO_IMAGE_NAME</name>
      <publish>true</publish>
    </image>
    <docker>
      <publishRegistry>
        <username>TODO_USERNAME</username>
        <password>TODO_PASSWORD</password>
      </publishRegistry>
    </docker>
  </configuration>
</plugin>
```

Provede publish (push)
image do Docker registry



Tady bohužel musí být v současnosti username & password :-(
Dá se to obejít tak, že se zadefinuje property např.:
 <docker.username>TODO</docker.username>
Poté se zde použije:
 <username>\${docker.username}</username>
A při spuštění se dá změnit z příkazové řádky:
mvn -Ddocker.username=TODO_REAL_VALUE spring-boot:build-image

GraalVM 21.3

- Podpora pro Java 17
- Odstranění podpory pro Java 8
- Od Spring Boot 2.6.0
- Od Spring Native 0.11.0 se ve výchozím nastavení nepoužívá GraalVM runtime, ale používá se NIK:
 - <https://bell-sw.com/pages/liberica-native-image-kit/>

GraalVM Dashboard & dive

- Abychom se podívali dovnitř vygenerované Native image, je možné použít tento nástroj:
 - <https://www.graalvm.org/dashboard/>
- Do konfigurace native-maven-plugin je nutné přidat:

```
<configuration>  
  <buildArgs>  
    <buildArg>-H:DashboardDump=dumpfileoversized</buildArg>  
    <buildArg>-H:+DashboardAll</buildArg>  
  </buildArgs>  
</configuration>
```

 - A pak spustit build pomocí `mvn -Pnative -DskipTests clean package`
- K tomu, abychom se podívali dovnitř vygenerované Docker image je možné použít aplikaci dive:
 - <https://github.com/wagoodman/dive>

GraalVM Community vs Enterprise

- GraalVM Community obsahuje pouze Serial GC (a Epsilon GC což je no-op GC). Enterprise verze obsahuje i G1:
 - Serial GC je vhodný pouze pro malý heap.
 - <https://www.graalvm.org/reference-manual/native-image/MemoryManagement/>

Heap size tuning

- Stačí spustit aplikaci tímto způsobem:
 - `docker run -m 200m --rm -it -p 8080:8080 IMG_NAME -XX:+PrintGC -XX:+VerboseGC`
- A do konzole se budou vypisovat informace při každém provedení GC

Další zajímavé odkazy

- Podporované části Spring Bootu:
 - <https://docs.spring.io/spring-native/docs/current/reference/htmlsingle/#support-spring-boot>
- Fungující příklady:
 - <https://github.com/spring-projects-experimental/spring-native/tree/main/samples>
- Build & Runtime argumenty:
 - https://www.graalvm.org/uploads/quick-references/native-image-quick-reference-v2_A4.pdf

Spring Native & produkce

- Základní otázka je, jestli používat Spring Native na produkci již nyní, nebo počkat až na Spring Boot 3 (konec roku 2022 – začátek roku 2023):
 - Pro Serverless nebo jednoduché microservices, kde není zapotřebí Spring Native složitě rozcházet a víceméně funguje out-of-the-box (případně s pár anotacemi), tak je Spring Native použitelný již nyní.
 - Pokud Vaše aplikace jednoduše fungovat nebude, tak nezoufejte:
 - Začátkem roku 2020 vytvořit hello world příklad se Spring Native trvalo půl dne a custom build skript. Po roce to samé byla otázka dvou příkazů.
 - Začátkem roku 2021 rozchodit relativně jednoduchou microservice co pracuje s databází se Spring Native vyžadovalo hromadu anotací a trvalo to cca. den. Dnes to funguje out-of-the-box.

Spring Native & produkce

- Problematické věci:
 - Metriky Actuatoru jsou hodně omezené, například neobsahují množství použité paměti
 - Problematický profiling. JFR (Java Flight Recorder) funguje omezeně:
 - <https://www.graalvm.org/reference-manual/native-image/JFR/>
- Moje pozorování z jednoduchých microservis, které jsem předělal do Native:
 - Aplikace, co na Hotspot JVM “žrala” 200MB RAM s OpenJ9 “žere” 100MB RAM a jako Native Image “žere” 50MB RAM.
 - Response time se překvapivě snížil o 50% (u jedné aplikace se snížil z 60 ms na 30 ms)
 - Start je cca. 70 ms (relativně jednoduchá stateless microservice, která pracuje s databází)
 - Build time je brutální (na mém 4 roky starém NTB 6 minut, na mém 4 roky starém desktopu 3 minuty) a když se ladí něco co nefunguje, tak to sežere hrozně moc času. Na druhou stranu to je ale dobrý důvod pro pořízení lepšího hardware ;-)

Děkuji za pozornost